JavaScript / EcmaScript 2017 API Reference

Table of contents

1. Introduction

This manual is intended to be used as a reference for Yoctopuce JavaScript / EcmaScript 2017 library, in order to interface your code with USB sensors and controllers.

You can use this JavaScript / EcmaScript 2017 library for your JavaScript applets that run within a web browser, as well as for your Node.js apps.

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 / EcmaScript

EcmaScript is the official name of the standardized version of the web-oriented programming language commonly referred to as *JavaScript*. This Yoctopuce library take advantages of advanced features introduced in EcmaScript 2017. It has therefore been named *Library for JavaScript / EcmaScript 2017* to differentiate it from the previous *Library for JavaScript*, now deprecated in favor of this new version.

This library provides access to Yoctopuce devices for modern JavaScript engines. It can be used within a browser as well as with Node.js. The library will automatically detect upon initialization whether the runtime environment is a browser or a Node.js virtual machine, and use the most appropriate system libraries accordingly.

Asynchronous communication with the devices is handled across the whole library using Promise objects, leveraging the new EcmaScript 2017 `async` / `await` non-blocking syntax for asynchronous I/O (see below). This syntax is now available out-of-the-box in most Javascript engines. No transpilation is needed: no Babel, no jspm, just plain Javascript. Here is your favorite engines minimum version needed to run this code. All of them are officially released at the time we write this document.

- Node.js v7.6 and later
- Firefox 52
- Opera 42 (incl. Android version)
- Chrome 55 (incl. Android version)
- Safari 10.1 (incl. iOS version)
- Android WebView 55
- Google V8 Javascript engine v5.5

If you need backward-compatibility with older releases, you can always run Babel to transpile your code and the library to older standards, as described a few paragraphs below.

We don't suggest using `jspm 0.17` anymore since that tool is still in Beta after 18 month, and having to use an extra tool to implement our library is pointless now that `async` / `await` are part of the standard.

2.1. Blocking I/O versus Asynchronous I/O in JavaScript

JavaScript is single-threaded by design. That means, if a program is actively waiting for the result of a network-based operation such as reading from a sensor, the whole program is blocked. In browser environments, this can even completely freeze the user interface. For this reason, the use of blocking I/O in JavaScript is strongly discouraged nowadays, and blocking network APIs are getting deprecated everywhere.

Instead of using parallel threads, JavaScript relies on asynchronous I/O to handle operations with a possible long timeout: whenever a long I/O call needs to be performed, it is only triggered and but then the code execution flow is terminated. The JavaScript engine is therefore free to handle other pending tasks, such as UI. Whenever the pending I/O call is completed, the system invokes a callback function with the result of the I/O call to resume execution of the original execution flow.

When used with plain callback functions, as pervasive in Node.js libraries, asynchronous I/O tend to produce code with poor readability, as the execution flow is broken into many disconnected callback functions. Fortunately, new methods have emerged recently to improve that situation. In particular, the use of *Promise* objects to abstract and work with asynchronous tasks helps a lot. Any function that makes a long I/O operation can return a *Promise*, which can be used by the caller to chain subsequent operations in the same flow. Promises are part of EcmaScript 2015 standard.

Promise objects are good, but what makes them even better is the new `async` / `await` keywords to handle asynchronous I/O:

- a function declared *async* will automatically encapsulate its result as a Promise
- within an *async* function, any function call prefixed with by *await* will chain the Promise returned by the function with a promise to resume execution of the caller
- any exception during the execution of an *async* function will automatically invoke the Promise failure continuation

Long story made short, *async* and *await* make it possible to write EcmaScript code with all benefits of asynchronous I/O, but without breaking the code flow. It is almost like multi-threaded execution, except that control switch between pending tasks only happens at places where the *await* keyword appears.

We have therefore chosen to write our new EcmaScript library using Promises and *async* functions, so that you can use the friendly *await* syntax. To keep it easy to remember, **all public methods** of the EcmaScript library **are *async***, i.e. return a Promise object, **except**:

- `GetTickCount()`, because returning a time stamp asynchronously does not make sense...
- `FindModule()`, `FirstModule()`, `nextModule()`,... because device detection and enumeration always work on internal device lists handled in background, and does not require immediate asynchronous I/O.

2.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

JavaScript is one of those languages which do not generally allow you to directly access the hardware layers of your computer. Therefore the library can only be used to access network-enabled devices (connected through a YoctoHub), or USB devices accessible through Yoctopuce TCP/IP to USB gateway, named *VirtualHub*.

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

- The Javascript / EcmaScript 2017 programming library[1]
- The VirtualHub software[2] for Windows, Mac OS X or Linux, depending on your OS

Extract the library files in a folder of your choice, you will find many of examples in it. Connect your modules and start the VirtualHub software. You do not need to install any driver.

Using the official Yoctopuce library for node.js

Start by installing the latest Node.js version (v7.6 or later) on your system. It is very easy. You can download it from the official web site: http://nodejs.org. Make sure to install it fully, including npm, and add it to the system path.

[1] www.yoctopuce.com/EN/libraries.php
[2] www.yoctopuce.com/EN/virtualhub.php

To give it a try, go into one of the example directory (for instance example_nodejs/Doc-Inventory). You will see that it include an application description file (package.json) and a source file (demo.js). To download and setup the libraries needed by this example, just run:

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

Using a local copy of the Yoctopuce library with node.js

If for some reason you need to make changes to the Yoctopuce library, you can easily configure your project to use the local copy in the `lib/` subdirectory rather than the official npm package. In order to do so, simply type the following command in your project directory:

```
npm link ../../lib
```

Using the Yoctopuce library within a browser (HTML)

For HTML examples, it is even simpler: there is nothing to install. Each example is a single HTML file that you can open in a browser to try it. In this context, loading the Yoctopuce library is no different from any standard HTML script include tag.

Using the Yoctoluce library on older JavaScript engines

If you need to run this library on older JavaScript engines, you can use Babel[3] to transpile your code and the library into older JavaScript standards. To install Babel with typical settings, simply use:

```
npm instal -g babel-cli
npm instal babel-preset-env
```

You would typically ask Babel to put the transpiled files in another directory, named `compat` for instance. Your files and all files of the Yoctopuce library should be transpiled, as follow:

```
babel --presets env demo.js --out-dir compat/
babel --presets env ../../lib --out-dir compat/
```

Although this approach is based on node.js toolchain, it actually works as well for transpiling JavaScript files for use in a browser. The only thing that you cannot do so easily is transpiling JavaScript code embedded directly in an HTML page. You have to use an external script file for using EcmaScript 2017 syntax with Babel.

Babel has many smart features, such as a watch mode that will automatically refresh transpiled files whenever the source file is changed, but this is beyond the scope of this note. You will find more in Babel documentation.

Backward-compatibility with the old JavaScript library

This new library is not fully backward-compatible with the old JavaScript library, because there is no way to transparently map the old blocking API to the new asynchronous API. The method names however are the same, and old synchronous code can easily be made asynchronous just by adding the proper `await` keywords before the method calls. For instance, simply replace:

```
beaconState = module.get_beacon();
```

by

[3] http://babeljs.io

```
beaconState = await module.get_beacon();
```

Apart from a few exceptions, most `XXX_async` redundant methods have been removed as well, as they would have introduced confusion on the proper way of handling asynchronous behaviors. It is however very simple to get an *async* method to invoke a callback upon completion, using the returned Promise object. For instance, you can replace:

```
module.get_beacon_async(callback, myContext);
```

by

```
module.get_beacon().then(function(res) { callback(myContext, module, res); });
```

In some cases, it might be desirable to get a sensor value using a method identical to the old synchronous methods (without using Promises), even if it returns a slightly outdated cached value since I/O is not possible. For this purpose, the EcmaScript library introduce new classes called *synchronous proxies*. A synchronous proxy is an object that mirrors the most recent state of the connected class, but can be read using regular synchronous function calls. For instance, instead of writing:

```
async function logInfo(module)
{
    console.log('Name: '+await module.get_logicalName());
    console.log('Beacon: '+await module.get_beacon());
}

...
logInfo(myModule);
...
```

you can use:

```
function logInfoProxy(moduleSyncProxy)
{
    console.log('Name: '+moduleProxy.get_logicalName());
    console.log('Beacon: '+moduleProxy.get_beacon());
}

logInfoSync(await myModule.get_syncProxy());
```

You can also rewrite this last asynchronous call as:

```
myModule.get_syncProxy().then(logInfoProxy);
```

2.3. Control of the Led function

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

```
// For Node.js, we use function require()
// For HTML, we would use &lt;script src="..."&gt;
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_led.js');

// Get access to your device, through the VirtualHub running locally
await YAPI.RegisterHub('127.0.0.1');
var led = YLed.FindLed("YCTOPOC1-123456.led1");

// Check that the module is online to handle hot-plug
if(await led.isOnline())
{
    // Use led.set_power()
    [...]
```

```
}
```

Let us look at these lines in more details.

yocto_api and yocto_led import

These two import provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_led` is necessary to manage modules containing a LED, such as Yocto-Demo. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

YAPI.RegisterHub

The `RegisterHub` method 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, or of a YoctoHub. If the host cannot be reached, this function will trigger an exception.

YLed.FindLed

The `FindLed` method 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 *led1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
led = YLed.FindLed("YCTOPOC1-123456.led1")
led = YLed.FindLed("YCTOPOC1-123456.MaFonction")
led = YLed.FindLed("MonModule.led1")
led = YLed.FindLed("MonModule.MaFonction")
led = YLed.FindLed("MaFonction")
```

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

isOnline

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

set_power

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

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-Demo** within Yoctopuce library for JavaScript / EcmaScript 2017. In there, you will find a file named `demo.js` with the sample code below, which uses the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

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.

```
"use strict";

require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_led.js');

async function startDemo(args)
{
```

```
    await YAPI.LogUnhandledPromiseRejections();
    await YAPI.DisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
        return;
    }

    // Select the relay to use
    let target;
    if(args[0] == "any") {
        let anyLed = YLed.FirstLed();
        if (anyLed == null) {
            console.log("No module connected (check USB cable)\n");
            process.exit(1);
        }
        let module = await anyLed.get_module();
        target = await module.get_serialNumber();
    } else {
        target = args[0];
    }

    // Switch relay as requested
    console.log("Turn LED " + args[1]);
    let led = YLed.FindLed(target + ".led");
    if(await led.isOnline()) {
        await led.set_power(args[1] == "ON" ? YLed.POWER_ON : YLed.POWER_OFF);
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }

    await YAPI.FreeAPI();
}

if(process.argv.length < 4) {
    console.log("usage: node demo.js <serial_number> [ ON | OFF ]");
    console.log("       node demo.js <logical_name> [ ON | OFF ]");
    console.log("       node demo.js any [ ON | OFF ]              (use any discovered
device)");
} else {
    startDemo(process.argv.slice(process.argv.length - 2));
}
```

As explained at the beginning of this chapter, you need to have Node.js v7.6 or later installed to try this example. When done, you can type the following two commands to automatically download and install the dependencies for building this example:

```
npm install
```

You can the start the sample code within Node.js using the following command, replacing the [...] by the arguments that you want to pass to the demo code:

```
node demo.js [...]
```

Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-Demo**. You will find there a single HTML file, with a JavaScript section similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_display.js"></script>
  <script>
```

```
    var led;

    async function startDemo()
    {
      await YAPI.LogUnhandledPromiseRejections();
      await YAPI.DisableExceptions();

      // Setup the API to use the VirtualHub on local machine
      let errmsg = new YErrorMsg();
      if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
      alert('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
    }
      refresh();
    }

    async function refresh()
    {
      let serial = document.getElementById('serial').value;
      if(serial == '') {
        // by default use any connected module suitable for the demo
        let anyLed = YLed.FirstLed();
        if(anyLed) {
          let module = await anyLed.module();
          serial = await module.get_serialNumber();
          document.getElementById('serial').value = serial;
        }
      }
      led = YLed.FindLed(serial+".led");
      if(await led.isOnline()) {
      document.getElementById('msg').value = '';
    } else {
      document.getElementById('msg').value = 'Module not connected';
    }
      setTimeout(refresh, 500);
    }

    window.sw = function(state)
    {
      led.set_power(state ? YLed.POWER_ON : YLed.POWER_OFF);
    };

    startDemo();
  </script>
</head>
<body>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>
 Turn LED <a href='javascript:sw(0);' >OFF</a> / <a href='javascript:sw(1);'>ON</a><br>
</body>
</html>
```

No installation is needed to run this example, all you have to do is open the HTML file using a web browser,

2.4. Control of the module part

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

```
"use strict";

require('yoctolib-es2017/yocto_api.js');

async function startDemo(args)
{
    await YAPI.LogUnhandledPromiseRejections();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
        return;
    }
```

```
    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            if(args[1] == 'ON') {
                await module.set_beacon(YModule.BEACON_ON);
            } else {
                await module.set_beacon(YModule.BEACON_OFF);
            }
        }
        console.log('serial:       '+await module.get_serialNumber());
        console.log('logical name: '+await module.get_logicalName());
        console.log('luminosity:   '+await module.get_luminosity()+'%');
        console.log('beacon:       '+(await module.get_beacon()==YModule.BEACON_ON
?'ON':'OFF'));
        console.log('upTime:       '+parseInt(await module.get_upTime()/1000)+' sec');
        console.log('USB current:  '+await module.get_usbCurrent()+' mA');
        console.log('logs:');
        console.log(await module.get_lastLogs());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

if(process.argv.length < 2) {
    console.log("usage: node demo.js <serial or logicalname> [ ON | OFF ]");
} else {
    startDemo(process.argv.slice(2));
}
```

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.

```
"use strict";

require('yoctolib-es2017/yocto_api.js');

async function startDemo(args)
{
    await YAPI.LogUnhandledPromiseRejections();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
        return;
    }

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            let newname = args[1];
            if (!await YAPI.CheckLogicalName(newname)) {
                console.log("Invalid name (" + newname + ")");
                process.exit(1);
            }
            await module.set_logicalName(newname);
            await module.saveToFlash();
        }
        console.log('Current name: '+await module.get_logicalName());
    } else {
```

```
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

if(process.argv.length < 2) {
    console.log("usage: node demo.js <serial> [newLogicalName]");
} else {
    startDemo(process.argv.slice(2));
}
```

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 `YModule.FirstModule()` 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.

```
"use strict";

require('yoctolib-es2017/yocto_api.js');

async function startDemo()
{
    await YAPI.LogUnhandledPromiseRejections();
    await YAPI.DisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if (await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1');
        return;
    }
    refresh();
}

async function refresh()
{
    try {
        let errmsg = new YErrorMsg();
        await YAPI.UpdateDeviceList(errmsg);

        let module = YModule.FirstModule();
        while(module) {
            let line = await module.get_serialNumber();
            line += '(' + (await module.get_productName()) + ')';
            console.log(line);
            module = module.nextModule();
        }
        setTimeout(refresh, 500);
    } catch(e) {
        console.log(e);
    }
}

try {
    startDemo();
} catch(e) {
    console.log(e);
}
```

2.5. Error handling

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

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

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

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

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

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

3. Reference

3.1. General functions

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

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

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');

Global functions
yCheckLogicalName(**name**) Checks if a given string is valid as logical name for a module or a function.
yClearHTTPCallbackCacheDir(**bool_removeFiles**) Disables the HTTP callback cache.
yDisableExceptions() Disables the use of exceptions to report runtime errors.
yEnableExceptions() Re-enables the use of exceptions for runtime error handling.
yEnableUSBHost(**osContext**) This function is used only on Android.
yFreeAPI() Frees dynamically allocated memory blocks used by the Yoctopuce library.
yGetAPIVersion() Returns the version identifier for the Yoctopuce library in use.
yGetTickCount() Returns the current value of a monotone millisecond-based time counter.
yHandleEvents(**errmsg**) Maintains the device-to-library communication channel.
yInitAPI(**mode**, **errmsg**) Initializes the Yoctopuce programming library explicitly.
yPreregisterHub(**url**, **errmsg**) Fault-tolerant alternative to RegisterHub().
yRegisterDeviceArrivalCallback(**arrivalCallback**) Register a callback function, to be called each time a device is plugged.
yRegisterDeviceRemovalCallback(**removalCallback**) Register a callback function, to be called each time a device is unplugged.

yRegisterHub(**url**, **errmsg**)

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

yRegisterHubDiscoveryCallback(**hubDiscoveryCallback**)

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

yRegisterLogFunction(**logfun**)

Registers a log callback function.

ySelectArchitecture(**arch**)

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

ySetDelegate(**object**)

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

ySetHTTPCallbackCacheDir(**str_directory**)

Enables the HTTP callback cache.

ySetTimeout(**callback**, **ms_timeout**, **args**)

Invoke the specified callback function after a given timeout.

ySetUSBPacketAckMs(**pktAckDelay**)

Enables the acknowledge of every USB packet received by the Yoctopuce library.

ySleep(**ms_duration**, **errmsg**)

Pauses the execution flow for a specified duration.

yTestHub(**url**, **mstimeout**, **errmsg**)

Test if the hub is reachable.

yTriggerHubDiscovery(**errmsg**)

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

yUnregisterHub(**url**)

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

yUpdateDeviceList(**errmsg**)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(**callback**, **context**)

Triggers a (re)detection of connected Yoctopuce modules.

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

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

function **CheckLogicalName**(**name**)

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

Parameters :

name a string containing the name to check.

Returns :

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

YAPI.DisableExceptions() YAPI yDisableExceptions()yDisableExceptions()

Disables the use of exceptions to report runtime errors.

```
function DisableExceptions( )
```

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

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

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

function **EnableExceptions**( )

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

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

Frees dynamically allocated memory blocks used by the Yoctopuce library.

function **FreeAPI**( )

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

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

Returns the version identifier for the Yoctopuce library in use.

function **GetAPIVersion**( )

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

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

Returns :
a character string describing the library version.

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

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

```
function GetTickCount( )
```

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

Returns :
a long integer corresponding to the millisecond counter.

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

Maintains the device-to-library communication channel.

function **HandleEvents**(**errmsg**)

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

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

Initializes the Yoctopuce programming library explicitly.

function **InitAPI**(**mode**, **errmsg**)

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

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

Fault-tolerant alternative to RegisterHub().

function **PreregisterHub**(**url**, **errmsg**)

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

Parameters :

url a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

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

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

function **RegisterDeviceArrivalCallback**(**arrivalCallback**)

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

Parameters :

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

YAPI.RegisterDeviceRemovalCallback() yRegisterDeviceRemovalCallback() yRegisterDeviceRemovalCallback()

YAPI

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

function **RegisterDeviceRemovalCallback**(**removalCallback**)

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

Parameters :

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

YAPI.RegisterHub() YAPI yRegisterHub()yRegisterHub()

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

function **RegisterHub**(**url**, **errmsg**)

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

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

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

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

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

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

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

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

Parameters :

url a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

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

Invoke the specified callback function after a given timeout.

function **SetTimeout**(**callback**, **ms_timeout**, **args**)

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.
args	additional arguments to be passed to the callback function can be provided, if needed (not supported on Microsoft Internet Explorer).

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

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

Pauses the execution flow for a specified duration.

function **Sleep**(**ms_duration**, **errmsg**)

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.TestHub() YAPI
yTestHub()

Test if the hub is reachable.

function **TestHub**(**url**, **mstimeout**)

This method do not register the hub, it only test if the hub is usable. The url parameter follow the same convention as the `RegisterHub` method. This method is useful to verify the authentication parameters for a hub. It is possible to force this method to return after mstimeout milliseconds.

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure returns a negative error code.

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

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

```
function UnregisterHub( url)
```

Parameters :

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

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

Triggers a (re)detection of connected Yoctopuce modules.

function **UpdateDeviceList**(**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.

3.2. Accelerometer function interface

The YSensor class is the parent class for all Yoctopuce sensors. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provide a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButton class is the only analog input which does not inherit from YSensor.

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

js	<script type='text/javascript' src='yocto_accelerometer.js'></script>
cpp	#include "yocto_accelerometer.h"
m	#import "yocto_accelerometer.h"
pas	uses yocto_accelerometer;
vb	yocto_accelerometer.vb
cs	yocto_accelerometer.cs
java	import com.yoctopuce.YoctoAPI.YAccelerometer;
uwp	import com.yoctopuce.YoctoAPI.YAccelerometer;
py	from yocto_accelerometer import *
php	require_once('yocto_accelerometer.php');
es	in HTML: <script src="../../lib/yocto_accelerometer.js"></script> in node.js: require('yoctolib-es2017/yocto_accelerometer.js');

Global functions

yFindAccelerometer(**func**)

Retrieves an accelerometer for a given identifier.

yFindAccelerometerInContext(**yctx**, **func**)

Retrieves an accelerometer for a given identifier in a YAPI context.

yFirstAccelerometer()

Starts the enumeration of accelerometers currently accessible.

yFirstAccelerometerInContext(**yctx**)

Starts the enumeration of accelerometers currently accessible.

`YAccelerometer` methods

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

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

accelerometer→clearCache()

Invalidates the cache.

accelerometer→describe()

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

accelerometer→get_advertisedValue()

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

accelerometer→get_bandwidth()

Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).

accelerometer→get_currentRawValue()

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

accelerometer→get_currentValue()

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

accelerometer→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

accelerometer→get_errorMessage()

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

accelerometer→get_errorType()

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

accelerometer→get_friendlyName()

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

accelerometer→get_functionDescriptor()

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

accelerometer→get_functionId()

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

accelerometer→get_hardwareId()

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

accelerometer→get_highestValue()

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

accelerometer→get_logFrequency()

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

accelerometer→get_logicalName()

Returns the logical name of the accelerometer.

accelerometer→get_lowestValue()

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

accelerometer→get_module()

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

accelerometer→get_module_async(**callback**, **context**)

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

accelerometer→get_recordedData(**startTime**, **endTime**)

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

accelerometer→get_reportFrequency()

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

accelerometer→get_resolution()

Returns the resolution of the measured values.

accelerometer→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

accelerometer→get_unit()

Returns the measuring unit for the acceleration.

accelerometer→get_userData()

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

accelerometer→get_xValue()

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

accelerometer→get_yValue()

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

accelerometer→get_zValue()

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

accelerometer→isOnline()

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

accelerometer→isOnline_async(**callback**, **context**)

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

accelerometer→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

accelerometer→load(**msValidity**)

Preloads the accelerometer cache with a specified validity duration.

accelerometer→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

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

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

accelerometer→load_async(**msValidity**, **callback**, **context**)

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

accelerometer→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

accelerometer→nextAccelerometer()

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

accelerometer→registerTimedReportCallback(**callback**)

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

accelerometer→registerValueCallback(**callback**)

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

accelerometer→set_bandwidth(**newval**)

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

accelerometer→set_highestValue(**newval**)

Changes the recorded maximal value observed.

accelerometer→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

accelerometer→set_logicalName(**newval**)

Changes the logical name of the accelerometer.

accelerometer→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

accelerometer→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

accelerometer→set_resolution(**newval**)

Changes the resolution of the measured physical values.

accelerometer→set_userData(**data**)

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

accelerometer→startDataLogger()

Starts the data logger on the device.

accelerometer→stopDataLogger()

Stops the datalogger on the device.

accelerometer→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

accelerometer→wait_async(callback, context)

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

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

Retrieves an accelerometer for a given identifier.

function **FindAccelerometer**(**func**)

The identifier can be specified using several formats:

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

This function does not require that the accelerometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAccelerometer.isOnline()` to test if the accelerometer is indeed online at a given time. In case of ambiguity when looking for an accelerometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the accelerometer

Returns :

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

YAccelerometer.FindAccelerometerInContext() YAccelerometer yFindAccelerometerInContext()

Retrieves an accelerometer for a given identifier in a YAPI context.

function **FindAccelerometerInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the accelerometer

Returns :

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

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

Starts the enumeration of accelerometers currently accessible.

function **FirstAccelerometer**( )

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

Returns :

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

YAccelerometer.FirstAccelerometerInContext() YAccelerometer yFirstAccelerometerInContext()

Starts the enumeration of accelerometers currently accessible.

function **FirstAccelerometerInContext**(**yctx**)

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

Parameters :

yctx a YAPI context.

Returns :

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

accelerometer→calibrateFromPoints() YAccelerometer
accelerometer.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.

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→clearCache() YAccelerometer
accelerometer.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the accelerometer attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

accelerometer→describe()accelerometer.describe() YAccelerometer

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

function **describe**( )

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

Returns :

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

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

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

function **get_advertisedValue**( )

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

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

accelerometer→get_bandwidth() YAccelerometer
accelerometer→bandwidth()
accelerometer.get_bandwidth()

Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **get_bandwidth**( )

Returns :
an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

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

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

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

function **get_currentRawValue**( )

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

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

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

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

function **get_currentValue**( )

Returns :

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

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

accelerometer→get_dataLogger() YAccelerometer
accelerometer→dataLogger()
accelerometer.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

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

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

function **get_errorMessage**( )

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

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

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

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

function **get_errorType**( )

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

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

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

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

function **get_friendlyName**( )

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

Returns :

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

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

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

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

function **get_functionDescriptor**( )

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

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

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

function **get_functionId**( )

For example `relay1`

Returns :

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

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

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

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

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the accelerometer (for example RELAYLO1-123456.relay1).

Returns :

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

accelerometer→get_highestValue() YAccelerometer
accelerometer→highestValue()
accelerometer.get_highestValue()

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

function **get_highestValue**( )

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

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

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

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

function **get_logFrequency**( )

Returns :

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

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

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

Returns the logical name of the accelerometer.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the accelerometer.

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

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

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

function **get_lowestValue**( )

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

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

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

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

function **get_module**( )

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

Returns :
an instance of YModule

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

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

function **get_recordedData**(**startTime**, **endTime**)

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

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

Parameters :

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

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

Returns :

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

accelerometer→get_reportFrequency() YAccelerometer
accelerometer→reportFrequency()
accelerometer.get_reportFrequency()

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

function **get_reportFrequency**( )

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

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

accelerometer→get_resolution() YAccelerometer
accelerometer→resolution()
accelerometer.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

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

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

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

accelerometer→get_sensorState() YAccelerometer
accelerometer→sensorState()
accelerometer.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

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

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

Returns the measuring unit for the acceleration.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the acceleration

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

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

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

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

function **get_xValue**( )

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

On failure, throws an exception or returns Y_XVALUE_INVALID.

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

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

function **get_yValue**( )

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

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

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

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

function **get_zValue**( )

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

On failure, throws an exception or returns Y_ZVALUE_INVALID.

accelerometer→isOnline()accelerometer.isOnline() YAccelerometer

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

function **isOnline**( )

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

Returns :

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

accelerometer→load()accelerometer.load() YAccelerometer

Preloads the accelerometer cache with a specified validity duration.

function **load**(**msValidity**)

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

accelerometer→loadAttribute() YAccelerometer
accelerometer.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

accelerometer→loadCalibrationPoints() YAccelerometer
accelerometer.loadCalibrationPoints()

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

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→muteValueCallbacks() YAccelerometer
accelerometer.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

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

accelerometer→nextAccelerometer() YAccelerometer
accelerometer.nextAccelerometer()

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

function **nextAccelerometer**( )

Returns :

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

accelerometer→registerTimedReportCallback() YAccelerometer
accelerometer.registerTimedReportCallback()

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

function **registerTimedReportCallback**(**callback**)

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

Parameters :

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

accelerometer→registerValueCallback() YAccelerometer
accelerometer.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.

accelerometer→set_bandwidth() YAccelerometer
accelerometer→setBandwidth()
accelerometer.set_bandwidth()

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

```
function set_bandwidth( newval)
```

When the frequency is lower, the device performs averaging.

Parameters :

newval an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

accelerometer→set_logFrequency() YAccelerometer
accelerometer→setLogFrequency()
accelerometer.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the logical name of the accelerometer.

function **set_logicalName**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

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

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_resolution() YAccelerometer
accelerometer→setResolution()
accelerometer.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→set_userData() YAccelerometer
accelerometer→setUserData()
accelerometer.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

accelerometer→startDataLogger() YAccelerometer
accelerometer.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

accelerometer→stopDataLogger() YAccelerometer
accelerometer.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

accelerometer→unmuteValueCallbacks() YAccelerometer
accelerometer.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

accelerometer→wait_async() YAccelerometer
accelerometer.wait_async()

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.3. Altitude function interface

The Yoctopuce class YAltitude allows you to read and configure Yoctopuce altitude sensors. It inherits from the YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to configure the barometric pressure adjusted to sea level (QNH) for barometric sensors.

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

js	<script type='text/javascript' src='yocto_altitude.js'></script>
cpp	#include "yocto_altitude.h"
m	#import "yocto_altitude.h"
pas	uses yocto_altitude;
vb	yocto_altitude.vb
cs	yocto_altitude.cs
java	import com.yoctopuce.YoctoAPI.YAltitude;
uwp	import com.yoctopuce.YoctoAPI.YAltitude;
py	from yocto_altitude import *
php	require_once('yocto_altitude.php');
es	in HTML: <script src="../../lib/yocto_altitude.js"></script> in node.js: require('yoctolib-es2017/yocto_altitude.js');

Global functions
yFindAltitude(**func**) Retrieves an altimeter for a given identifier.
yFindAltitudeInContext(**yctx**, **func**) Retrieves an altimeter for a given identifier in a YAPI context.
yFirstAltitude() Starts the enumeration of altimeters currently accessible.
yFirstAltitudeInContext(**yctx**) Starts the enumeration of altimeters currently accessible.
YAltitude methods
altitude→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
altitude→clearCache() Invalidates the cache.
altitude→describe() Returns a short text that describes unambiguously the instance of the altimeter in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
altitude→get_advertisedValue() Returns the current value of the altimeter (no more than 6 characters).
altitude→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in meters, as a floating point number.
altitude→get_currentValue() Returns the current value of the altitude, in meters, as a floating point number.
altitude→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
altitude→get_errorMessage() Returns the error message of the latest error with the altimeter.

altitude→get_errorType()

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

altitude→get_friendlyName()

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

altitude→get_functionDescriptor()

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

altitude→get_functionId()

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

altitude→get_hardwareId()

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

altitude→get_highestValue()

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

altitude→get_logFrequency()

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

altitude→get_logicalName()

Returns the logical name of the altimeter.

altitude→get_lowestValue()

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

altitude→get_module()

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

altitude→get_module_async(**callback**, **context**)

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

altitude→get_qnh()

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

altitude→get_recordedData(**startTime**, **endTime**)

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

altitude→get_reportFrequency()

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

altitude→get_resolution()

Returns the resolution of the measured values.

altitude→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

altitude→get_technology()

Returns the technology used by the sesnor to compute altitude.

altitude→get_unit()

Returns the measuring unit for the altitude.

altitude→get_userData()

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

altitude→isOnline()

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

altitude→isOnline_async(**callback**, **context**)

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

altitude→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

altitude→load(msValidity)

Preloads the altimeter cache with a specified validity duration.

altitude→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

altitude→loadCalibrationPoints(rawValues, refValues)

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

altitude→load_async(msValidity, callback, context)

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

altitude→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

altitude→nextAltitude()

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

altitude→registerTimedReportCallback(callback)

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

altitude→registerValueCallback(callback)

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

altitude→set_currentValue(newval)

Changes the current estimated altitude.

altitude→set_highestValue(newval)

Changes the recorded maximal value observed.

altitude→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

altitude→set_logicalName(newval)

Changes the logical name of the altimeter.

altitude→set_lowestValue(newval)

Changes the recorded minimal value observed.

altitude→set_qnh(newval)

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

altitude→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

altitude→set_resolution(newval)

Changes the resolution of the measured physical values.

altitude→set_userData(data)

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

altitude→startDataLogger()

Starts the data logger on the device.

altitude→stopDataLogger()

Stops the datalogger on the device.

altitude→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

altitude→wait_async(callback, context)

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

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

Retrieves an altimeter for a given identifier.

function **FindAltitude**(**func**)

The identifier can be specified using several formats:

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

This function does not require that the altimeter is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAltitude.isOnline()` to test if the altimeter is indeed online at a given time. In case of ambiguity when looking for an altimeter by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the altimeter

Returns :

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

YAltitude.FindAltitudeInContext() YAltitude yFindAltitudeInContext()

Retrieves an altimeter for a given identifier in a YAPI context.

function **FindAltitudeInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the altimeter

Returns :

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

YAltitude.FirstAltitude() YAltitude yFirstAltitude()yFirstAltitude()

Starts the enumeration of altimeters currently accessible.

function **FirstAltitude**( )

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

Returns :

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

YAltitude.FirstAltitudeInContext() YAltitude yFirstAltitudeInContext()

Starts the enumeration of altimeters currently accessible.

```
function FirstAltitudeInContext( yctx)
```

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

Parameters :

yctx a YAPI context.

Returns :

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

altitude→calibrateFromPoints() YAltitude
altitude.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.

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→clearCache()altitude.clearCache() YAltitude

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the altimeter attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

altitude→describe()altitude.describe() YAltitude

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

function **describe**( )

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

Returns :

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

altitude→get_advertisedValue() YAltitude
altitude→advertisedValue()
altitude.get_advertisedValue()

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

function **get_advertisedValue**( )

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

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

altitude→get_currentRawValue() YAltitude
altitude→currentRawValue()
altitude.get_currentRawValue()

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

function **get_currentRawValue**( )

Returns :

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

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

altitude→get_currentValue() YAltitude
altitude→currentValue()altitude.get_currentValue()

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

function **get_currentValue**( )

Returns :

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

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

altitude→get_dataLogger() YAltitude
altitude→dataLogger()altitude.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

altitude→get_errorMessage() YAltitude
altitude→errorMessage()altitude.get_errorMessage()

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

function **get_errorMessage**( )

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

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

altitude→get_errorType() YAltitude
altitude→errorType()altitude.get_errorType()

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

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 the altimeter object

altitude→get_friendlyName() YAltitude
altitude→friendlyName()altitude.get_friendlyName()

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

function **get_friendlyName**( )

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

altitude→get_functionDescriptor() YAltitude
altitude→functionDescriptor()
altitude.get_functionDescriptor()

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.

altitude→get_functionId() YAltitude
altitude→functionId()altitude.get_functionId()

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

function **get_functionId**( )

For example `relay1`

Returns :

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

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

altitude→get_hardwareId() YAltitude
altitude→hardwareId()altitude.get_hardwareId()

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

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the altimeter (for example RELAYLO1-123456.relay1).

Returns :

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

altitude→get_highestValue() YAltitude
altitude→highestValue()altitude.get_highestValue()

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

function **get_highestValue**( )

Returns :

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

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

altitude→get_logFrequency() YAltitude
altitude→logFrequency()altitude.get_logFrequency()

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

function **get_logFrequency**( )

Returns :

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

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

altitude→get_logicalName() YAltitude
altitude→logicalName()altitude.get_logicalName()

Returns the logical name of the altimeter.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the altimeter.

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

altitude→get_lowestValue() YAltitude
altitude→lowestValue()altitude.get_lowestValue()

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

function **get_lowestValue**( )

Returns :

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

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

altitude→get_module() YAltitude
altitude→module()altitude.get_module()

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

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

altitude→get_qnh() YAltitude
altitude→qnh()altitude.get_qnh()

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

function **get_qnh**( )

Returns :

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

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

altitude→get_recordedData() YAltitude
altitude→recordedData()altitude.get_recordedData()

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

function **get_recordedData**(**startTime**, **endTime**)

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

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

Parameters :

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

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

Returns :

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

altitude→get_reportFrequency() YAltitude
altitude→reportFrequency()
altitude.get_reportFrequency()

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

function **get_reportFrequency**( )

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

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

altitude→get_resolution() YAltitude
altitude→resolution()altitude.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

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

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

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

altitude→get_sensorState() YAltitude
altitude→sensorState()altitude.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

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

altitude→get_technology() YAltitude
altitude→technology()altitude.get_technology()

Returns the technology used by the sesnor to compute altitude.

```
function get_technology( )
```

Possibles values are "barometric" and "gps"

Returns :
a string corresponding to the technology used by the sesnor to compute altitude

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

altitude→get_unit() YAltitude
altitude→unit()altitude.get_unit()

Returns the measuring unit for the altitude.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the altitude

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

altitude→get_userData() YAltitude
altitude→userData()altitude.get_userData()

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

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.

altitude→isOnline()altitude.isOnline() YAltitude

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

function **isOnline**( )

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

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

altitude→load()altitude.load() YAltitude

Preloads the altimeter cache with a specified validity duration.

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.

altitude→loadAttribute()altitude.loadAttribute() YAltitude

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

altitude→loadCalibrationPoints() YAltitude
altitude.loadCalibrationPoints()

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

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→muteValueCallbacks() YAltitude
altitude.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

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

altitude→nextAltitude()altitude.nextAltitude() YAltitude

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

function **nextAltitude**( )

Returns :

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

altitude→registerTimedReportCallback() YAltitude
altitude.registerTimedReportCallback()

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

function **registerTimedReportCallback**(**callback**)

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

Parameters :

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

altitude→registerValueCallback() YAltitude
altitude.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.

altitude→set_currentValue() YAltitude
altitude→setCurrentValue()
altitude.set_currentValue()

Changes the current estimated altitude.

function **set_currentValue**(**newval**)

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

Parameters :

newval a floating point number corresponding to the current estimated altitude

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_highestValue() YAltitude
altitude→setHighestValue()
altitude.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.

altitude→set_logFrequency() YAltitude
altitude→setLogFrequency()
altitude.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→set_logicalName() YAltitude
altitude→setLogicalName()altitude.set_logicalName()

Changes the logical name of the altimeter.

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 altimeter.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→set_lowestValue() YAltitude
altitude→setLowestValue()altitude.set_lowestValue()

Changes the recorded minimal value observed.

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.

altitude→set_qnh() YAltitude
altitude→setQnh()altitude.set_qnh()

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

function **set_qnh**(**newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_reportFrequency() YAltitude
altitude→setReportFrequency()
altitude.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_resolution() YAltitude
altitude→setResolution()altitude.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_userData() YAltitude
altitude→setUserData()altitude.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

altitude→startDataLogger()altitude.startDataLogger() YAltitude

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

altitude→stopDataLogger()altitude.stopDataLogger() YAltitude

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

altitude→unmuteValueCallbacks() YAltitude
altitude.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

altitude→wait_async()altitude.wait_async() YAltitude

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.4. AnButton function interface

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

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

```
js      <script type='text/javascript' src='yocto_anbutton.js'></script>
cpp     #include "yocto_anbutton.h"
m       #import "yocto_anbutton.h"
pas     uses yocto_anbutton;
vb      yocto_anbutton.vb
cs      yocto_anbutton.cs
java    import com.yoctopuce.YoctoAPI.YAnButton;
uwp     import com.yoctopuce.YoctoAPI.YAnButton;
py      from yocto_anbutton import *
php     require_once('yocto_anbutton.php');
es      in HTML: <script src="../../lib/yocto_anbutton.js"></script>
        in node.js: require('yoctolib-es2017/yocto_anbutton.js');
```

Global functions
yFindAnButton(**func**) Retrieves an analog input for a given identifier.
yFindAnButtonInContext(**yctx**, **func**) Retrieves an analog input for a given identifier in a YAPI context.
yFirstAnButton() Starts the enumeration of analog inputs currently accessible.
yFirstAnButtonInContext(**yctx**) Starts the enumeration of analog inputs currently accessible.
YAnButton methods
anbutton→clearCache() Invalidates the cache.
anbutton→describe() Returns a short text that describes unambiguously the instance of the analog input in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
anbutton→get_advertisedValue() Returns the current value of the analog input (no more than 6 characters).
anbutton→get_analogCalibration() Tells if a calibration process is currently ongoing.
anbutton→get_calibratedValue() Returns the current calibrated input value (between 0 and 1000, included).
anbutton→get_calibrationMax() Returns the maximal value measured during the calibration (between 0 and 4095, included).
anbutton→get_calibrationMin() Returns the minimal value measured during the calibration (between 0 and 4095, included).
anbutton→get_errorMessage()

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

anbutton→get_errorType()

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

anbutton→get_friendlyName()

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

anbutton→get_functionDescriptor()

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

anbutton→get_functionId()

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

anbutton→get_hardwareId()

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

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

anbutton→get_module_async(**callback**, **context**)

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

anbutton→get_pulseCounter()

Returns the pulse counter value.

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms).

anbutton→get_rawValue()

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

anbutton→get_sensitivity()

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

anbutton→get_userData()

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

anbutton→isOnline()

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

anbutton→isOnline_async(**callback**, **context**)

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

anbutton→load(**msValidity**)

Preloads the analog input cache with a specified validity duration.

anbutton→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

anbutton→load_async(**msValidity**, **callback**, **context**)

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

anbutton→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

anbutton→nextAnButton()

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

anbutton→registerValueCallback(**callback**)

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

anbutton→resetCounter()

Returns the pulse counter value as well as its timer.

anbutton→set_analogCalibration(**newval**)

Starts or stops the calibration process.

anbutton→set_calibrationMax(**newval**)

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

anbutton→set_calibrationMin(**newval**)

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

anbutton→set_logicalName(**newval**)

Changes the logical name of the analog input.

anbutton→set_sensitivity(**newval**)

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

anbutton→set_userData(**data**)

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

anbutton→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

anbutton→wait_async(**callback**, **context**)

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

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

Retrieves an analog input for a given identifier.

```
function FindAnButton( func)
```

The identifier can be specified using several formats:

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

This function does not require that the analog input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAnButton.isOnline()` to test if the analog input is indeed online at a given time. In case of ambiguity when looking for an analog input by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the analog input

Returns :

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

YAnButton.FindAnButtonInContext() YAnButton yFindAnButtonInContext()

Retrieves an analog input for a given identifier in a YAPI context.

function **FindAnButtonInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the analog input

Returns :

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

YAnButton.FirstAnButton() YAnButton yFirstAnButton()yFirstAnButton()

Starts the enumeration of analog inputs currently accessible.

function **FirstAnButton**( )

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

Returns :

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

YAnButton.FirstAnButtonInContext() YAnButton yFirstAnButtonInContext()

Starts the enumeration of analog inputs currently accessible.

function **FirstAnButtonInContext**(**yctx**)

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

Parameters :

yctx a YAPI context.

Returns :

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

anbutton→clearCache()anbutton.clearCache() YAnButton

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the analog input attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

anbutton→describe()anbutton.describe() YAnButton

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

function **describe**( )

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

Returns :

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

anbutton→get_advertisedValue() YAnButton
anbutton→advertisedValue()
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() YAnButton
anbutton→analogCalibration()
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() YAnButton
anbutton→calibratedValue()
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() YAnButton
anbutton→calibrationMax()
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() YAnButton
anbutton→calibrationMin()
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() YAnButton
anbutton→errorMessage()
anbutton.get_errorMessage()

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

function **get_errorMessage**( )

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

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

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

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

function **get_errorType**( )

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

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

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

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

function **get_friendlyName**( )

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

Returns :

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

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

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

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

function **get_functionDescriptor**( )

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

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

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

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the analog input (ex: `relay1`)

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

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

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

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the analog input (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the analog input (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

anbutton→get_isPressed() YAnButton
anbutton→isPressed()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() YAnButton
anbutton→lastTimePressed()
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 transitioned from open to closed)

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

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

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

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 transitioned from closed to open)

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

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

Returns the logical name of the analog input.

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

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

function **get_module**( )

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

Returns :
an instance of YModule

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

Returns the pulse counter value.

function **get_pulseCounter**( )

The value is a 32 bit integer. In case of overflow (>=2^32), the counter will wrap. To reset the counter, just call the resetCounter() method.

Returns :
an integer corresponding to the pulse counter value

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

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

Returns the timer of the pulses counter (ms).

function **get_pulseTimer**( )

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

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

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

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

function **get_sensitivity**( )

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

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

anbutton→get_userData() YAnButton
anbutton→userData()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()anbutton.isOnline() YAnButton

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

function **isOnline**( )

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

Returns :

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

anbutton→load()anbutton.load() YAnButton

Preloads the analog input cache with a specified validity duration.

function **load**(**msValidity**)

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

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

anbutton→muteValueCallbacks() YAnButton
anbutton.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

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

anbutton→nextAnButton()anbutton.nextAnButton() YAnButton

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

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() YAnButton
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→resetCounter()anbutton.resetCounter() YAnButton

Returns the pulse counter value as well as its timer.

function **resetCounter**( )

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→set_analogCalibration() YAnButton
anbutton→setAnalogCalibration()
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() YAnButton
anbutton→setCalibrationMax()
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() YAnButton
anbutton→setCalibrationMin()
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() YAnButton
anbutton→setLogicalName()
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() YAnButton
anbutton→setSensitivity()anbutton.set_sensitivity()

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

function **set_sensitivity**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton→set_userData() YAnButton
anbutton→setUserData()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

anbutton→unmuteValueCallbacks() YAnButton
anbutton.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

anbutton→wait_async()anbutton.wait_async() YAnButton

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.5. AudioIn function interface

The Yoctopuce application programming interface allows you to configure the volume of the input channel.

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

js	<script type='text/javascript' src='yocto_audioin.js'></script>
cpp	#include "yocto_audioin.h"
m	#import "yocto_audioin.h"
pas	uses yocto_audioin;
vb	yocto_audioin.vb
cs	yocto_audioin.cs
java	import com.yoctopuce.YoctoAPI.YAudioIn;
uwp	import com.yoctopuce.YoctoAPI.YAudioIn;
py	from yocto_audioin import *
php	require_once('yocto_audioin.php');
es	in HTML: <script src="../../lib/yocto_audioin.js"></script> in node.js: require('yoctolib-es2017/yocto_audioin.js');

Global functions

yFindAudioIn(**func**)

Retrieves an audio input for a given identifier.

yFindAudioInInContext(**yctx**, **func**)

Retrieves an audio input for a given identifier in a YAPI context.

yFirstAudioIn()

Starts the enumeration of audio inputs currently accessible.

yFirstAudioInInContext(**yctx**)

Starts the enumeration of audio inputs currently accessible.

YAudioIn methods

audioin→clearCache()

Invalidates the cache.

audioin→describe()

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

audioin→get_advertisedValue()

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

audioin→get_errorMessage()

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

audioin→get_errorType()

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

audioin→get_friendlyName()

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

audioin→get_functionDescriptor()

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

audioin→get_functionId()

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

audioin→get_hardwareId()

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

audioin→get_logicalName()
Returns the logical name of the audio input.

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

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

audioin→get_mute()
Returns the state of the mute function.

audioin→get_noSignalFor()
Returns the number of seconds elapsed without detecting a signal.

audioin→get_signal()
Returns the detected input signal level.

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

audioin→get_volume()
Returns audio input gain, in per cents.

audioin→get_volumeRange()
Returns the supported volume range.

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

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

audioin→load(**msValidity**)
Preloads the audio input cache with a specified validity duration.

audioin→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

audioin→load_async(**msValidity**, **callback**, **context**)
Preloads the audio input cache with a specified validity duration (asynchronous version).

audioin→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

audioin→nextAudioIn()
Continues the enumeration of audio inputs started using `yFirstAudioIn()`.

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

audioin→set_logicalName(**newval**)
Changes the logical name of the audio input.

audioin→set_mute(**newval**)
Changes the state of the mute function.

audioin→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

audioin→set_volume(**newval**)
Changes audio input gain, in per cents.

audioin→unmuteValueCallbacks()
Re-enables the propagation of every new advertised value to the parent hub.

audioin→wait_async(**callback**, **context**)

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

YAudioIn.FindAudioIn() YAudioIn yFindAudioIn()yFindAudioIn()

Retrieves an audio input for a given identifier.

```
function FindAudioIn( 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 audio input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAudioIn.isOnline()` to test if the audio input is indeed online at a given time. In case of ambiguity when looking for an audio 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the audio input

Returns :

a `YAudioIn` object allowing you to drive the audio input.

YAudioIn.FindAudioInInContext() YAudioIn yFindAudioInInContext()

Retrieves an audio input for a given identifier in a YAPI context.

function **FindAudioInInContext**(**yctx**, **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 audio input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAudioIn.isOnline()` to test if the audio input is indeed online at a given time. In case of ambiguity when looking for an audio 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 :

yctx a YAPI context

func a string that uniquely characterizes the audio input

Returns :

a `YAudioIn` object allowing you to drive the audio input.

YAudioIn.FirstAudioIn() YAudioIn yFirstAudioIn()yFirstAudioIn()

Starts the enumeration of audio inputs currently accessible.

function **FirstAudioIn**( )

Use the method `YAudioIn.nextAudioIn()` to iterate on next audio inputs.

Returns :

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

YAudioIn.FirstAudioInInContext() YAudioIn yFirstAudioInInContext()

Starts the enumeration of audio inputs currently accessible.

function **FirstAudioInInContext**(**yctx**)

Use the method `YAudioIn.nextAudioIn()` to iterate on next audio inputs.

Parameters :

yctx a YAPI context.

Returns :

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

audioin→clearCache()audioin.clearCache() YAudioIn

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the audio input attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

audioin→describe()audioin.describe() YAudioIn

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

function **describe**( )

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

Returns :

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

audioin→get_advertisedValue() YAudioIn
audioin→advertisedValue()
audioin.get_advertisedValue()

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

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the audio input (no more than 6 characters).

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

audioin→get_errorMessage() YAudioIn
audioin→errorMessage()audioin.get_errorMessage()

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

function **get_errorMessage**( )

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

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

audioin→get_errorType() YAudioIn
audioin→errorType()audioin.get_errorType()

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

function **get_errorType**( )

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

Returns :
a number corresponding to the code of the latest error that occurred while using the audio input object

audioin→get_friendlyName() YAudioIn
audioin→friendlyName()audioin.get_friendlyName()

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

function **get_friendlyName**( )

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

audioin→get_functionDescriptor() YAudioIn
audioin→functionDescriptor()
audioin.get_functionDescriptor()

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.

audioin→get_functionId() YAudioIn
audioin→functionId()audioin.get_functionId()

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

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the audio input (ex: `relay1`)

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

audioin→get_hardwareId() YAudioIn
audioin→hardwareId()audioin.get_hardwareId()

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

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the audio input (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the audio input (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

audioin→get_logicalName() YAudioIn
audioin→logicalName()audioin.get_logicalName()

Returns the logical name of the audio input.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the audio input.

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

audioin→get_module() YAudioIn
audioin→module()audioin.get_module()

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

function **get_module**( )

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

Returns :
an instance of YModule

audioin→get_mute() YAudioIn
audioin→mute()audioin.get_mute()

Returns the state of the mute function.

function **get_mute**( )

Returns :

either Y_MUTE_FALSE or Y_MUTE_TRUE, according to the state of the mute function

On failure, throws an exception or returns Y_MUTE_INVALID.

audioin→get_noSignalFor() YAudioIn
audioin→noSignalFor()audioin.get_noSignalFor()

Returns the number of seconds elapsed without detecting a signal.

function **get_noSignalFor**( )

Returns :
an integer corresponding to the number of seconds elapsed without detecting a signal

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

audioin→get_signal() YAudioIn
audioin→signal()audioin.get_signal()

Returns the detected input signal level.

function **get_signal**( )

Returns :
an integer corresponding to the detected input signal level

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

audioin→get_userData() YAudioIn
audioin→userData()audioin.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.

audioin→get_volume() YAudioIn
audioin→volume()audioin.get_volume()

Returns audio input gain, in per cents.

function **get_volume**( )

Returns :
an integer corresponding to audio input gain, in per cents

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

audioin→get_volumeRange() YAudioIn
audioin→volumeRange()audioin.get_volumeRange()

Returns the supported volume range.

function **get_volumeRange**( )

The low value of the range corresponds to the minimal audible value. To completely mute the sound, use `set_mute()` instead of the `set_volume()`.

Returns :
a string corresponding to the supported volume range

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

audioin→isOnline()audioin.isOnline() YAudioIn

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

function **isOnline**( )

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

Returns :

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

audioin→load()audioin.load() YAudioIn

Preloads the audio input cache with a specified validity duration.

function **load**(**msValidity**)

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

audioin→loadAttribute()audioin.loadAttribute() YAudioIn

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

audioin→muteValueCallbacks() YAudioIn
audioin.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

audioin→nextAudioIn()audioin.nextAudioIn() YAudioIn

Continues the enumeration of audio inputs started using `yFirstAudioIn()`.

function **nextAudioIn**( )

Returns :

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

audioin→registerValueCallback() YAudioIn
audioin.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.

audioin→set_logicalName() YAudioIn
audioin→setLogicalName()audioin.set_logicalName()

Changes the logical name of the audio 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 audio input.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

audioin→set_mute() YAudioIn
audioin→setMute()audioin.set_mute()

Changes the state of the mute function.

```
function set_mute( newval)
```

Remember to call the matching module `saveToFlash()` method to save the setting permanently.

Parameters :

newval either `Y_MUTE_FALSE` or `Y_MUTE_TRUE`, according to the state of the mute function

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

audioin→set_userData() YAudioIn
audioin→setUserData()audioin.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

audioin→set_volume() YAudioIn
audioin→setVolume()audioin.set_volume()

Changes audio input gain, in per cents.

```
function set_volume( newval)
```

Parameters :

newval an integer corresponding to audio input gain, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

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

audioin→unmuteValueCallbacks() YAudioIn
audioin.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

audioin→wait_async()audioin.wait_async() YAudioIn

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.6. AudioOut function interface

The Yoctopuce application programming interface allows you to configure the volume of the outout.

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

js	<script type='text/javascript' src='yocto_audioout.js'></script>
cpp	#include "yocto_audioout.h"
m	#import "yocto_audioout.h"
pas	uses yocto_audioout;
vb	yocto_audioout.vb
cs	yocto_audioout.cs
java	import com.yoctopuce.YoctoAPI.YAudioOut;
uwp	import com.yoctopuce.YoctoAPI.YAudioOut;
py	from yocto_audioout import *
php	require_once('yocto_audioout.php');
es	in HTML: <script src="../../lib/yocto_audioout.js"></script> in node.js: require('yoctolib-es2017/yocto_audioout.js');

Global functions

yFindAudioOut(**func**)

Retrieves an audio output for a given identifier.

yFindAudioOutInContext(**yctx**, **func**)

Retrieves an audio output for a given identifier in a YAPI context.

yFirstAudioOut()

Starts the enumeration of audio outputs currently accessible.

yFirstAudioOutInContext(**yctx**)

Starts the enumeration of audio outputs currently accessible.

YAudioOut methods

audioout→clearCache()

Invalidates the cache.

audioout→describe()

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

audioout→get_advertisedValue()

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

audioout→get_errorMessage()

Returns the error message of the latest error with the audio output.

audioout→get_errorType()

Returns the numerical error code of the latest error with the audio output.

audioout→get_friendlyName()

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

audioout→get_functionDescriptor()

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

audioout→get_functionId()

Returns the hardware identifier of the audio output, without reference to the module.

audioout→get_hardwareId()

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

audioout→get_logicalName()

Returns the logical name of the audio output.

audioout→get_module()

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

audioout→get_module_async(**callback**, **context**)

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

audioout→get_mute()

Returns the state of the mute function.

audioout→get_noSignalFor()

Returns the number of seconds elapsed without detecting a signal.

audioout→get_signal()

Returns the detected output current level.

audioout→get_userData()

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

audioout→get_volume()

Returns audio output volume, in per cents.

audioout→get_volumeRange()

Returns the supported volume range.

audioout→isOnline()

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

audioout→isOnline_async(**callback**, **context**)

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

audioout→load(**msValidity**)

Preloads the audio output cache with a specified validity duration.

audioout→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

audioout→load_async(**msValidity**, **callback**, **context**)

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

audioout→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

audioout→nextAudioOut()

Continues the enumeration of audio outputs started using `yFirstAudioOut()`.

audioout→registerValueCallback(**callback**)

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

audioout→set_logicalName(**newval**)

Changes the logical name of the audio output.

audioout→set_mute(**newval**)

Changes the state of the mute function.

audioout→set_userData(**data**)

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

audioout→set_volume(**newval**)

Changes audio output volume, in per cents.

audioout→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

audioout→wait_async(**callback**, **context**)

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

YAudioOut.FindAudioOut() YAudioOut yFindAudioOut()yFindAudioOut()

Retrieves an audio output for a given identifier.

```
function FindAudioOut( 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 audio output is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAudioOut.isOnline()` to test if the audio output is indeed online at a given time. In case of ambiguity when looking for an audio output 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the audio output

Returns :

a `YAudioOut` object allowing you to drive the audio output.

YAudioOut.FindAudioOutInContext() YAudioOut yFindAudioOutInContext()

Retrieves an audio output for a given identifier in a YAPI context.

function **FindAudioOutInContext**(**yctx**, **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 audio output is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAudioOut.isOnline()` to test if the audio output is indeed online at a given time. In case of ambiguity when looking for an audio output 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 :

yctx a YAPI context

func a string that uniquely characterizes the audio output

Returns :

a `YAudioOut` object allowing you to drive the audio output.

YAudioOut.FirstAudioOut() YAudioOut yFirstAudioOut()yFirstAudioOut()

Starts the enumeration of audio outputs currently accessible.

function **FirstAudioOut**( )

Use the method `YAudioOut.nextAudioOut()` to iterate on next audio outputs.

Returns :

a pointer to a `YAudioOut` object, corresponding to the first audio output currently online, or a `null` pointer if there are none.

YAudioOut.FirstAudioOutInContext() YAudioOut yFirstAudioOutInContext()

Starts the enumeration of audio outputs currently accessible.

function **FirstAudioOutInContext**(**yctx**)

Use the method `YAudioOut.nextAudioOut()` to iterate on next audio outputs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YAudioOut` object, corresponding to the first audio output currently online, or a `null` pointer if there are none.

audioout→clearCache()audioout.clearCache() YAudioOut

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the audio output attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

audioout→describe()audioout.describe() YAudioOut

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

function **describe**( )

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

Returns :

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

audioout→get_advertisedValue() YAudioOut
audioout→advertisedValue()
audioout.get_advertisedValue()

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

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the audio output (no more than 6 characters).

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

audioout→get_errorMessage() YAudioOut
audioout→errorMessage()
audioout.get_errorMessage()

Returns the error message of the latest error with the audio output.

function **get_errorMessage**( )

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

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

audioout→get_errorType() YAudioOut
audioout→errorType()audioout.get_errorType()

Returns the numerical error code of the latest error with the audio output.

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 the audio output object

audioout→get_friendlyName() YAudioOut
audioout→friendlyName()
audioout.get_friendlyName()

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

function **get_friendlyName**( )

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

Returns :

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

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

audioout→get_functionDescriptor() YAudioOut
audioout→functionDescriptor()
audioout.get_functionDescriptor()

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.

audioout→get_functionId() YAudioOut
audioout→functionId()audioout.get_functionId()

Returns the hardware identifier of the audio output, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the audio output (ex: `relay1`)

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

audioout→get_hardwareId() YAudioOut
audioout→hardwareId()audioout.get_hardwareId()

Returns the unique hardware identifier of the audio output in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the audio output (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the audio output (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

audioout→get_logicalName() YAudioOut
audioout→logicalName()audioout.get_logicalName()

Returns the logical name of the audio output.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the audio output.

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

audioout→get_module() YAudioOut
audioout→module()audioout.get_module()

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

function **get_module**( )

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

Returns :
an instance of YModule

audioout→get_mute() YAudioOut
audioout→mute()audioout.get_mute()

Returns the state of the mute function.

function **get_mute**( )

Returns :

either Y_MUTE_FALSE or Y_MUTE_TRUE, according to the state of the mute function

On failure, throws an exception or returns Y_MUTE_INVALID.

audioout→get_noSignalFor() YAudioOut
audioout→noSignalFor()audioout.get_noSignalFor()

Returns the number of seconds elapsed without detecting a signal.

function **get_noSignalFor**( )

Returns :
an integer corresponding to the number of seconds elapsed without detecting a signal

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

audioout→get_signal() YAudioOut
audioout→signal()audioout.get_signal()

Returns the detected output current level.

function **get_signal**( )

Returns :
an integer corresponding to the detected output current level

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

audioout→get_userData() YAudioOut
audioout→userData()audioout.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.

audioout→get_volume() YAudioOut
audioout→volume()audioout.get_volume()

Returns audio output volume, in per cents.

function **get_volume**( )

Returns :
an integer corresponding to audio output volume, in per cents

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

audioout→get_volumeRange() YAudioOut
audioout→volumeRange()
audioout.get_volumeRange()

Returns the supported volume range.

function **get_volumeRange**( )

The low value of the range corresponds to the minimal audible value. To completely mute the sound, use `set_mute()` instead of the `set_volume()`.

Returns :
a string corresponding to the supported volume range

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

audioout→isOnline()audioout.isOnline() YAudioOut

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

function **isOnline**( )

If there is a cached value for the audio output 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 audio output.

Returns :

`true` if the audio output can be reached, and `false` otherwise

audioout→load()audioout.load() YAudioOut

Preloads the audio output 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.

audioout→loadAttribute()audioout.loadAttribute() YAudioOut

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

audioout→muteValueCallbacks() YAudioOut
audioout.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

audioout→nextAudioOut()audioout.nextAudioOut() YAudioOut

Continues the enumeration of audio outputs started using `yFirstAudioOut()`.

function **nextAudioOut**( )

Returns :

a pointer to a `YAudioOut` object, corresponding to an audio output currently online, or a `null` pointer if there are no more audio outputs to enumerate.

audioout→registerValueCallback() YAudioOut
audioout.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.

audioout→set_logicalName() YAudioOut
audioout→setLogicalName()
audioout.set_logicalName()

Changes the logical name of the audio output.

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 audio output.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

audioout→set_mute() YAudioOut
audioout→setMute()audioout.set_mute()

Changes the state of the mute function.

function **set_mute**(**newval**)

Remember to call the matching module `saveToFlash()` method to save the setting permanently.

Parameters :

newval either `Y_MUTE_FALSE` or `Y_MUTE_TRUE`, according to the state of the mute function

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

audioout→set_userData() YAudioOut
audioout→setUserData()audioout.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

audioout→set_volume() YAudioOut
audioout→setVolume()audioout.set_volume()

Changes audio output volume, in per cents.

function **set_volume**(**newval**)

Parameters :

newval an integer corresponding to audio output volume, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

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

audioout→unmuteValueCallbacks() YAudioOut
audioout.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

audioout→wait_async()audioout.wait_async() YAudioOut

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.7. BluetoothLink function interface

BluetoothLink function provides control over bluetooth link and status for devices that are bluetooth-enabled.

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

```
js      <script type='text/javascript' src='yocto_bluetoothlink.js'></script>
cpp     #include "yocto_bluetoothlink.h"
m       #import "yocto_bluetoothlink.h"
pas     uses yocto_bluetoothlink;
vb      yocto_bluetoothlink.vb
cs      yocto_bluetoothlink.cs
java    import com.yoctopuce.YoctoAPI.YBluetoothLink;
uwp     import com.yoctopuce.YoctoAPI.YBluetoothLink;
py      from yocto_bluetoothlink import *
php     require_once('yocto_bluetoothlink.php');
es      in HTML: <script src="../../lib/yocto_bluetoothlink.js"></script>
        in node.js: require('yoctolib-es2017/yocto_bluetoothlink.js');
```

Global functions
yFindBluetoothLink(**func**) Retrieves a cellular interface for a given identifier.
yFindBluetoothLinkInContext(**yctx**, **func**) Retrieves a cellular interface for a given identifier in a YAPI context.
yFirstBluetoothLink() Starts the enumeration of cellular interfaces currently accessible.
yFirstBluetoothLinkInContext(**yctx**) Starts the enumeration of cellular interfaces currently accessible.
`YBluetoothLink` methods
bluetoothlink→clearCache() Invalidates the cache.
bluetoothlink→connect() Attempt to connect to the previously selected remote device.
bluetoothlink→describe() Returns a short text that describes unambiguously the instance of the cellular interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
bluetoothlink→disconnect() Disconnect from the previously selected remote device.
bluetoothlink→get_advertisedValue() Returns the current value of the cellular interface (no more than 6 characters).
bluetoothlink→get_errorMessage() Returns the error message of the latest error with the cellular interface.
bluetoothlink→get_errorType() Returns the numerical error code of the latest error with the cellular interface.
bluetoothlink→get_friendlyName() Returns a global identifier of the cellular interface in the format `MODULE_NAME.FUNCTION_NAME`.
bluetoothlink→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

bluetoothlink→get_functionId()

Returns the hardware identifier of the cellular interface, without reference to the module.

bluetoothlink→get_hardwareId()

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

bluetoothlink→get_linkQuality()

Returns the bluetooth receiver signal strength, in pourcents, or 0 if no connection is established.

bluetoothlink→get_linkState()

Returns the bluetooth link state.

bluetoothlink→get_logicalName()

Returns the logical name of the cellular interface.

bluetoothlink→get_module()

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

bluetoothlink→get_module_async(callback, context)

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

bluetoothlink→get_mute()

Returns the state of the mute function.

bluetoothlink→get_ownAddress()

Returns the MAC-48 address of the bluetooth interface, which is unique on the bluetooth network.

bluetoothlink→get_pairingPin()

Returns an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card.

bluetoothlink→get_preAmplifier()

Returns the audio pre-amplifier volume, in per cents.

bluetoothlink→get_remoteAddress()

Returns the MAC-48 address of the remote device to connect to.

bluetoothlink→get_remoteName()

Returns the bluetooth name the remote device, if found on the bluetooth network.

bluetoothlink→get_userData()

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

bluetoothlink→get_volume()

Returns the connected headset volume, in per cents.

bluetoothlink→isOnline()

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

bluetoothlink→isOnline_async(callback, context)

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

bluetoothlink→load(msValidity)

Preloads the cellular interface cache with a specified validity duration.

bluetoothlink→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

bluetoothlink→load_async(msValidity, callback, context)

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

bluetoothlink→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

bluetoothlink→nextBluetoothLink()

Continues the enumeration of cellular interfaces started using `yFirstBluetoothLink()`.

bluetoothlink→registerValueCallback(callback)

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

bluetoothlink→set_logicalName(newval)

Changes the logical name of the cellular interface.

bluetoothlink→set_mute(newval)

Changes the state of the mute function.

bluetoothlink→set_pairingPin(newval)

Changes the PIN code used by the module for bluetooth pairing.

bluetoothlink→set_preAmplifier(newval)

Changes the audio pre-amplifier volume, in per cents.

bluetoothlink→set_remoteAddress(newval)

Changes the MAC-48 address defining which remote device to connect to.

bluetoothlink→set_userData(data)

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

bluetoothlink→set_volume(newval)

Changes the connected headset volume, in per cents.

bluetoothlink→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

bluetoothlink→wait_async(callback, context)

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

YBluetoothLink.FindBluetoothLink() YBluetoothLink yFindBluetoothLink()yFindBluetoothLink()

Retrieves a cellular interface for a given identifier.

```
function FindBluetoothLink( 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 cellular interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YBluetoothLink.isOnline()` to test if the cellular interface is indeed online at a given time. In case of ambiguity when looking for a cellular 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the cellular interface

Returns :

a `YBluetoothLink` object allowing you to drive the cellular interface.

YBluetoothLink.FindBluetoothLinkInContext() yFindBluetoothLinkInContext()

YBluetoothLink

Retrieves a cellular interface for a given identifier in a YAPI context.

function **FindBluetoothLinkInContext**(**yctx**, **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 cellular interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YBluetoothLink.isOnline()` to test if the cellular interface is indeed online at a given time. In case of ambiguity when looking for a cellular 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 :

yctx a YAPI context

func a string that uniquely characterizes the cellular interface

Returns :

a `YBluetoothLink` object allowing you to drive the cellular interface.

YBluetoothLink.FirstBluetoothLink() YBluetoothLink yFirstBluetoothLink()yFirstBluetoothLink()

Starts the enumeration of cellular interfaces currently accessible.

function **FirstBluetoothLink**( )

Use the method `YBluetoothLink.nextBluetoothLink()` to iterate on next cellular interfaces.

Returns :

a pointer to a `YBluetoothLink` object, corresponding to the first cellular interface currently online, or a `null` pointer if there are none.

YBluetoothLink.FirstBluetoothLinkInContext() YBluetoothLink yFirstBluetoothLinkInContext()

Starts the enumeration of cellular interfaces currently accessible.

function **FirstBluetoothLinkInContext**(**yctx**)

Use the method `YBluetoothLink.nextBluetoothLink()` to iterate on next cellular interfaces.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YBluetoothLink` object, corresponding to the first cellular interface currently online, or a `null` pointer if there are none.

bluetoothlink→clearCache() YBluetoothLink
bluetoothlink.clearCache()

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the cellular interface attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

bluetoothlink→connect()bluetoothlink.connect() YBluetoothLink

Attempt to connect to the previously selected remote device.

function **connect**( )

Returns :

YAPI_SUCCESS when the call succeeds.

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

bluetoothlink→describe()bluetoothlink.describe() YBluetoothLink

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

function **describe**( )

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

Returns :

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

bluetoothlink→disconnect() YBluetoothLink
bluetoothlink.disconnect()

Disconnect from the previously selected remote device.

function **disconnect**( )

Returns :

YAPI_SUCCESS when the call succeeds.

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

bluetoothlink→get_advertisedValue() YBluetoothLink
bluetoothlink→advertisedValue()
bluetoothlink.get_advertisedValue()

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

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the cellular interface (no more than 6 characters).

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

bluetoothlink→get_errorMessage() YBluetoothLink
bluetoothlink→errorMessage()
bluetoothlink.get_errorMessage()

Returns the error message of the latest error with the cellular interface.

function **get_errorMessage**( )

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

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

bluetoothlink→get_errorType() YBluetoothLink
bluetoothlink→errorType()
bluetoothlink.get_errorType()

Returns the numerical error code of the latest error with the cellular interface.

function **get_errorType**( )

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

Returns :
a number corresponding to the code of the latest error that occurred while using the cellular interface object

bluetoothlink→get_friendlyName() YBluetoothLink
bluetoothlink→friendlyName()
bluetoothlink.get_friendlyName()

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

function **get_friendlyName**( )

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

Returns :

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

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

bluetoothlink→get_functionDescriptor() YBluetoothLink
bluetoothlink→functionDescriptor()
bluetoothlink.get_functionDescriptor()

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.

bluetoothlink→get_functionId() YBluetoothLink
bluetoothlink→functionId()
bluetoothlink.get_functionId()

Returns the hardware identifier of the cellular interface, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the cellular interface (ex: `relay1`)

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

bluetoothlink→get_hardwareId() YBluetoothLink
bluetoothlink→hardwareId()
bluetoothlink.get_hardwareId()

Returns the unique hardware identifier of the cellular interface in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the cellular interface (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the cellular interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

bluetoothlink→get_linkQuality() YBluetoothLink
bluetoothlink→linkQuality()
bluetoothlink.get_linkQuality()

Returns the bluetooth receiver signal strength, in pourcents, or 0 if no connection is established.

function **get_linkQuality**( )

Returns :
an integer corresponding to the bluetooth receiver signal strength, in pourcents, or 0 if no connection is established

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

bluetoothlink→get_linkState() YBluetoothLink
bluetoothlink→linkState()
bluetoothlink.get_linkState()

Returns the bluetooth link state.

function **get_linkState**( )

Returns :

a value among `Y_LINKSTATE_DOWN`, `Y_LINKSTATE_FREE`, `Y_LINKSTATE_SEARCH`, `Y_LINKSTATE_EXISTS`, `Y_LINKSTATE_LINKED` and `Y_LINKSTATE_PLAY` corresponding to the bluetooth link state

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

bluetoothlink→get_logicalName() YBluetoothLink
bluetoothlink→logicalName()
bluetoothlink.get_logicalName()

Returns the logical name of the cellular interface.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the cellular interface.

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

bluetoothlink→get_module() YBluetoothLink
bluetoothlink→module()bluetoothlink.get_module()

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

function **get_module**( )

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

Returns :
an instance of YModule

bluetoothlink→get_mute() YBluetoothLink
bluetoothlink→mute()bluetoothlink.get_mute()

Returns the state of the mute function.

function **get_mute**( )

Returns :

either Y_MUTE_FALSE or Y_MUTE_TRUE, according to the state of the mute function

On failure, throws an exception or returns Y_MUTE_INVALID.

bluetoothlink→get_ownAddress() YBluetoothLink
bluetoothlink→ownAddress()
bluetoothlink.get_ownAddress()

Returns the MAC-48 address of the bluetooth interface, which is unique on the bluetooth network.

function **get_ownAddress**( )

Returns :

a string corresponding to the MAC-48 address of the bluetooth interface, which is unique on the bluetooth network

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

bluetoothlink→get_pairingPin() YBluetoothLink
bluetoothlink→pairingPin()
bluetoothlink.get_pairingPin()

Returns an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card.

function **get_pairingPin**( )

Returns :

a string corresponding to an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card

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

bluetoothlink→get_preAmplifier() YBluetoothLink
bluetoothlink→preAmplifier()
bluetoothlink.get_preAmplifier()

Returns the audio pre-amplifier volume, in per cents.

function **get_preAmplifier**( )

Returns :
an integer corresponding to the audio pre-amplifier volume, in per cents

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

bluetoothlink→get_remoteAddress() YBluetoothLink
bluetoothlink→remoteAddress()
bluetoothlink.get_remoteAddress()

Returns the MAC-48 address of the remote device to connect to.

function **get_remoteAddress**( )

Returns :
a string corresponding to the MAC-48 address of the remote device to connect to

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

bluetoothlink→get_remoteName() YBluetoothLink
bluetoothlink→remoteName()
bluetoothlink.get_remoteName()

Returns the bluetooth name the remote device, if found on the bluetooth network.

function **get_remoteName**( )

Returns :
a string corresponding to the bluetooth name the remote device, if found on the bluetooth network

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

bluetoothlink→get_userData() YBluetoothLink
bluetoothlink→userData()
bluetoothlink.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.

bluetoothlink→get_volume() YBluetoothLink
bluetoothlink→volume()bluetoothlink.get_volume()

Returns the connected headset volume, in per cents.

function **get_volume**( )

Returns :
an integer corresponding to the connected headset volume, in per cents

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

bluetoothlink→isOnline()bluetoothlink.isOnline() YBluetoothLink

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

function **isOnline**( )

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

Returns :

`true` if the cellular interface can be reached, and `false` otherwise

bluetoothlink→load()bluetoothlink.load() YBluetoothLink

Preloads the cellular interface cache with a specified validity duration.

function **load**(**msValidity**)

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

bluetoothlink→loadAttribute() YBluetoothLink
bluetoothlink.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

bluetoothlink→muteValueCallbacks() YBluetoothLink
bluetoothlink.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

bluetoothlink→nextBluetoothLink() YBluetoothLink
bluetoothlink.nextBluetoothLink()

Continues the enumeration of cellular interfaces started using `yFirstBluetoothLink()`.

function **nextBluetoothLink**( )

Returns :

a pointer to a `YBluetoothLink` object, corresponding to a cellular interface currently online, or a `null` pointer if there are no more cellular interfaces to enumerate.

bluetoothlink→registerValueCallback() YBluetoothLink
bluetoothlink.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.

bluetoothlink→set_logicalName() YBluetoothLink
bluetoothlink→setLogicalName()
bluetoothlink.set_logicalName()

Changes the logical name of the cellular 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 cellular interface.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

bluetoothlink→set_mute() YBluetoothLink
bluetoothlink→setMute()bluetoothlink.set_mute()

Changes the state of the mute function.

function **set_mute**(**newval**)

Remember to call the matching module `saveToFlash()` method to save the setting permanently.

Parameters :

newval either `Y_MUTE_FALSE` or `Y_MUTE_TRUE`, according to the state of the mute function

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

bluetoothlink→set_pairingPin() YBluetoothLink
bluetoothlink→setPairingPin()
bluetoothlink.set_pairingPin()

Changes the PIN code used by the module for bluetooth pairing.

function **set_pairingPin**(**newval**)

Remember to call the `saveToFlash()` method of the module to save the new value in the device flash.

Parameters :

newval a string corresponding to the PIN code used by the module for bluetooth pairing

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

bluetoothlink→set_preAmplifier() YBluetoothLink
bluetoothlink→setPreAmplifier()
bluetoothlink.set_preAmplifier()

Changes the audio pre-amplifier volume, in per cents.

function **set_preAmplifier**(**newval**)

Parameters :

newval an integer corresponding to the audio pre-amplifier volume, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

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

bluetoothlink→set_remoteAddress() YBluetoothLink
bluetoothlink→setRemoteAddress()
bluetoothlink.set_remoteAddress()

Changes the MAC-48 address defining which remote device to connect to.

function **set_remoteAddress**(**newval**)

Parameters :

newval a string corresponding to the MAC-48 address defining which remote device to connect to

Returns :

YAPI_SUCCESS if the call succeeds.

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

bluetoothlink→set_userData() YBluetoothLink
bluetoothlink→setUserData()
bluetoothlink.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

bluetoothlink→set_volume()
bluetoothlink→setVolume()
bluetoothlink.set_volume()

YBluetoothLink

Changes the connected headset volume, in per cents.

function **set_volume**(**newval**)

Parameters :

newval an integer corresponding to the connected headset volume, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

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

bluetoothlink→unmuteValueCallbacks() YBluetoothLink
bluetoothlink.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

bluetoothlink→wait_async() YBluetoothLink
bluetoothlink.wait_async()

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.8. Buzzer function interface

The Yoctopuce application programming interface allows you to choose the frequency and volume at which the buzzer must sound. You can also pre-program a play sequence.

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

js	<script type='text/javascript' src='yocto_buzzer.js'></script>
cpp	#include "yocto_buzzer.h"
m	#import "yocto_buzzer.h"
pas	uses yocto_buzzer;
vb	yocto_buzzer.vb
cs	yocto_buzzer.cs
java	import com.yoctopuce.YoctoAPI.YBuzzer;
uwp	import com.yoctopuce.YoctoAPI.YBuzzer;
py	from yocto_buzzer import *
php	require_once('yocto_buzzer.php');
es	in HTML: <script src="../../lib/yocto_buzzer.js"></script> in node.js: require('yoctolib-es2017/yocto_buzzer.js');

Global functions
yFindBuzzer(**func**) Retrieves a buzzer for a given identifier.
yFindBuzzerInContext(**yctx**, **func**) Retrieves a buzzer for a given identifier in a YAPI context.
yFirstBuzzer() Starts the enumeration of buzzers currently accessible.
yFirstBuzzerInContext(**yctx**) Starts the enumeration of buzzers currently accessible.
YBuzzer methods
buzzer→addFreqMoveToPlaySeq(**freq**, **msDelay**) Adds a new frequency transition to the playing sequence.
buzzer→addNotesToPlaySeq(**notes**) Adds notes to the playing sequence.
buzzer→addPulseToPlaySeq(**freq**, **msDuration**) Adds a pulse to the playing sequence.
buzzer→addVolMoveToPlaySeq(**volume**, **msDuration**) Adds a new volume transition to the playing sequence.
buzzer→clearCache() Invalidates the cache.
buzzer→describe() Returns a short text that describes unambiguously the instance of the buzzer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
buzzer→freqMove(**frequency**, **duration**) Makes the buzzer frequency change over a period of time.
buzzer→get_advertisedValue() Returns the current value of the buzzer (no more than 6 characters).
buzzer→get_errorMessage() Returns the error message of the latest error with the buzzer.

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

buzzer→get_frequency()
Returns the frequency of the signal sent to the buzzer/speaker.

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

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

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

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

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

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

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

buzzer→get_playSeqMaxSize()
Returns the maximum length of the playing sequence.

buzzer→get_playSeqSignature()
Returns the playing sequence signature.

buzzer→get_playSeqSize()
Returns the current length of the playing sequence.

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

buzzer→get_volume()
Returns the volume of the signal sent to the buzzer/speaker.

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

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

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

buzzer→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

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

buzzer→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

buzzer→nextBuzzer()
Continues the enumeration of buzzers started using `yFirstBuzzer()`.

buzzer→oncePlaySeq()
Starts the preprogrammed playing sequence and run it once only.

buzzer→playNotes(notes)

Immediately play a note sequence.

buzzer→pulse(frequency, duration)

Activates the buzzer for a short duration.

buzzer→registerValueCallback(callback)

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

buzzer→resetPlaySeq()

Resets the preprogrammed playing sequence and sets the frequency to zero.

buzzer→set_frequency(newval)

Changes the frequency of the signal sent to the buzzer.

buzzer→set_logicalName(newval)

Changes the logical name of the buzzer.

buzzer→set_userData(data)

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

buzzer→set_volume(newval)

Changes the volume of the signal sent to the buzzer/speaker.

buzzer→startPlaySeq()

Starts the preprogrammed playing sequence.

buzzer→stopPlaySeq()

Stops the preprogrammed playing sequence and sets the frequency to zero.

buzzer→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

buzzer→volumeMove(volume, duration)

Makes the buzzer volume change over a period of time, frequency stays untouched.

buzzer→wait_async(callback, context)

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

YBuzzer.FindBuzzer() YBuzzer yFindBuzzer()yFindBuzzer()

Retrieves a buzzer for a given identifier.

```
function FindBuzzer( 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 buzzer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YBuzzer.isOnline()` to test if the buzzer is indeed online at a given time. In case of ambiguity when looking for a buzzer 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the buzzer

Returns :

a `YBuzzer` object allowing you to drive the buzzer.

YBuzzer.FindBuzzerInContext() YBuzzer yFindBuzzerInContext()

Retrieves a buzzer for a given identifier in a YAPI context.

function **FindBuzzerInContext**(**yctx**, **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 buzzer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YBuzzer.isOnline()` to test if the buzzer is indeed online at a given time. In case of ambiguity when looking for a buzzer 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 :

yctx a YAPI context

func a string that uniquely characterizes the buzzer

Returns :

a `YBuzzer` object allowing you to drive the buzzer.

YBuzzer.FirstBuzzer() YBuzzer yFirstBuzzer()yFirstBuzzer()

Starts the enumeration of buzzers currently accessible.

function **FirstBuzzer**( )

Use the method YBuzzer.nextBuzzer() to iterate on next buzzers.

Returns :

a pointer to a YBuzzer object, corresponding to the first buzzer currently online, or a null pointer if there are none.

YBuzzer.FirstBuzzerInContext() YBuzzer yFirstBuzzerInContext()

Starts the enumeration of buzzers currently accessible.

```
function FirstBuzzerInContext( yctx)
```

Use the method `YBuzzer.nextBuzzer()` to iterate on next buzzers.

Parameters :

yctx a YAPI context.

Returns :

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

buzzer→addFreqMoveToPlaySeq() YBuzzer
buzzer.addFreqMoveToPlaySeq()

Adds a new frequency transition to the playing sequence.

function **addFreqMoveToPlaySeq**(**freq**, **msDelay**)

Parameters :

freq desired frequency when the transition is completed, in Hz

msDelay duration of the frequency transition, in milliseconds.

Returns :

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

buzzer→addNotesToPlaySeq() YBuzzer
buzzer.addNotesToPlaySeq()

Adds notes to the playing sequence.

function **addNotesToPlaySeq**(**notes**)

Notes are provided as text words, separated by spaces. The pitch is specified using the usual letter from A to G. The duration is specified as the divisor of a whole note: 4 for a fourth, 8 for an eight note, etc. Some modifiers are supported: # and b to alter a note pitch, ' and , to move to the upper/lower octave, . to enlarge the note duration.

Parameters :

notes notes to be played, as a text string.

Returns :

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

buzzer→addPulseToPlaySeq() YBuzzer
buzzer.addPulseToPlaySeq()

Adds a pulse to the playing sequence.

function **addPulseToPlaySeq**(**freq**, **msDuration**)

Parameters :

freq	pulse frequency, in Hz
msDuration	pulse duration, in milliseconds.

Returns :

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

buzzer→addVolMoveToPlaySeq() YBuzzer
buzzer.addVolMoveToPlaySeq()

Adds a new volume transition to the playing sequence.

function **addVolMoveToPlaySeq**(**volume**, **msDuration**)

Frequency stays untouched: if frequency is at zero, the transition has no effect.

Parameters :

volume	desired volume when the transition is completed, as a percentage.
msDuration	duration of the volume transition, in milliseconds.

Returns :

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

buzzer→clearCache()buzzer.clearCache() YBuzzer

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the buzzer attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

buzzer→describe()buzzer.describe() YBuzzer

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

function **describe**( )

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

Returns :

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

buzzer→freqMove()buzzer.freqMove() YBuzzer

Makes the buzzer frequency change over a period of time.

function **freqMove**(**frequency**, **duration**)

Parameters :

frequency frequency to reach, in hertz. A frequency under 25Hz stops the buzzer.

duration pulse duration in millseconds

Returns :

YAPI_SUCCESS if the call succeeds.

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

buzzer→get_advertisedValue() YBuzzer
buzzer→advertisedValue()
buzzer.get_advertisedValue()

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

function **get_advertisedValue**( )

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

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

buzzer→get_errorMessage() YBuzzer
buzzer→errorMessage()buzzer.get_errorMessage()

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

function **get_errorMessage**( )

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

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

buzzer→get_errorType() YBuzzer
buzzer→errorType()buzzer.get_errorType()

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

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 the buzzer object

buzzer→get_frequency() YBuzzer
buzzer→frequency()buzzer.get_frequency()

Returns the frequency of the signal sent to the buzzer/speaker.

function **get_frequency**( )

Returns :
a floating point number corresponding to the frequency of the signal sent to the buzzer/speaker

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

buzzer→get_friendlyName() YBuzzer
buzzer→friendlyName()buzzer.get_friendlyName()

Returns a global identifier of the buzzer in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

buzzer→get_functionDescriptor() YBuzzer
buzzer→functionDescriptor()
buzzer.get_functionDescriptor()

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.

buzzer→get_functionId() YBuzzer
buzzer→functionId()buzzer.get_functionId()

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

function **get_functionId**( )

For example `relay1`

Returns :

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

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

buzzer→get_hardwareId() YBuzzer
buzzer→hardwareId()buzzer.get_hardwareId()

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

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the buzzer (for example RELAYLO1-123456.relay1).

Returns :

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

buzzer→get_logicalName() YBuzzer
buzzer→logicalName()buzzer.get_logicalName()

Returns the logical name of the buzzer.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the buzzer.

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

buzzer→get_module() YBuzzer
buzzer→module()buzzer.get_module()

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

function **get_module**( )

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

Returns :
an instance of YModule

buzzer→get_playSeqMaxSize() YBuzzer
buzzer→playSeqMaxSize()
buzzer.get_playSeqMaxSize()

Returns the maximum length of the playing sequence.

function **get_playSeqMaxSize**( )

Returns :
an integer corresponding to the maximum length of the playing sequence

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

buzzer→get_playSeqSignature() YBuzzer
buzzer→playSeqSignature()
buzzer.get_playSeqSignature()

Returns the playing sequence signature.

function **get_playSeqSignature**( )

As playing sequences cannot be read from the device, this can be used to detect if a specific playing sequence is already programmed.

Returns :
an integer corresponding to the playing sequence signature

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

buzzer→get_playSeqSize() YBuzzer
buzzer→playSeqSize()buzzer.get_playSeqSize()

Returns the current length of the playing sequence.

function **get_playSeqSize**( )

Returns :
an integer corresponding to the current length of the playing sequence

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

buzzer→get_userData() YBuzzer
buzzer→userData()buzzer.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.

buzzer→get_volume() YBuzzer
buzzer→volume()buzzer.get_volume()

Returns the volume of the signal sent to the buzzer/speaker.

function **get_volume**( )

Returns :
an integer corresponding to the volume of the signal sent to the buzzer/speaker

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

buzzer→isOnline()buzzer.isOnline() YBuzzer

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

function **isOnline**( )

If there is a cached value for the buzzer 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 buzzer.

Returns :

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

buzzer→load()buzzer.load() YBuzzer

Preloads the buzzer 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.

buzzer→loadAttribute()buzzer.loadAttribute() YBuzzer

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

buzzer→muteValueCallbacks() YBuzzer
buzzer.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

buzzer→nextBuzzer()buzzer.nextBuzzer() YBuzzer

Continues the enumeration of buzzers started using `yFirstBuzzer()`.

function **nextBuzzer**( )

Returns :

a pointer to a `YBuzzer` object, corresponding to a buzzer currently online, or a `null` pointer if there are no more buzzers to enumerate.

buzzer→oncePlaySeq()buzzer.oncePlaySeq() YBuzzer

Starts the preprogrammed playing sequence and run it once only.

function **oncePlaySeq**( )

Returns :

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

buzzer→playNotes()buzzer.playNotes() YBuzzer

Immediately play a note sequence.

function **playNotes**(**notes**)

Notes are provided as text words, separated by spaces. The pitch is specified using the usual letter from A to G. The duration is specified as the divisor of a whole note: 4 for a fourth, 8 for an eight note, etc. Some modifiers are supported: # and b to alter a note pitch, ' and , to move to the upper/lower octave, . to enlarge the note duration.

Parameters :

notes notes to be played, as a text string.

Returns :

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

buzzer→pulse()buzzer.pulse() YBuzzer

Activates the buzzer for a short duration.

function **pulse**(**frequency**, **duration**)

Parameters :

frequency pulse frequency, in hertz

duration pulse duration in millseconds

Returns :

YAPI_SUCCESS if the call succeeds.

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

buzzer→registerValueCallback() YBuzzer
buzzer.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.

buzzer→resetPlaySeq()buzzer.resetPlaySeq() YBuzzer

Resets the preprogrammed playing sequence and sets the frequency to zero.

function **resetPlaySeq**( )

Returns :

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

buzzer→set_frequency() YBuzzer
buzzer→setFrequency()buzzer.set_frequency()

Changes the frequency of the signal sent to the buzzer.

function **set_frequency**(**newval**)

A zero value stops the buzzer.

Parameters :

newval a floating point number corresponding to the frequency of the signal sent to the buzzer

Returns :

YAPI_SUCCESS if the call succeeds.

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

buzzer→set_logicalName() YBuzzer
buzzer→setLogicalName()buzzer.set_logicalName()

Changes the logical name of the buzzer.

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 buzzer.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

buzzer→set_userData() YBuzzer
buzzer→setUserData()buzzer.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

buzzer→set_volume() YBuzzer
buzzer→setVolume()buzzer.set_volume()

Changes the volume of the signal sent to the buzzer/speaker.

function **set_volume**(**newval**)

Parameters :

newval an integer corresponding to the volume of the signal sent to the buzzer/speaker

Returns :

YAPI_SUCCESS if the call succeeds.

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

buzzer→startPlaySeq()buzzer.startPlaySeq() YBuzzer

Starts the preprogrammed playing sequence.

function **startPlaySeq**( )

The sequence runs in loop until it is stopped by stopPlaySeq or an explicit change. To play the sequence only once, use `oncePlaySeq()`.

Returns :

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

buzzer→stopPlaySeq()buzzer.stopPlaySeq() YBuzzer

Stops the preprogrammed playing sequence and sets the frequency to zero.

function **stopPlaySeq**( )

Returns :

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

buzzer→unmuteValueCallbacks() YBuzzer
buzzer.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

buzzer→volumeMove()buzzer.volumeMove() YBuzzer

Makes the buzzer volume change over a period of time, frequency stays untouched.

function **volumeMove**(**volume**, **duration**)

Parameters :

volume volume to reach in %

duration change duration in millseconds

Returns :

YAPI_SUCCESS if the call succeeds.

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

buzzer→wait_async()buzzer.wait_async() YBuzzer

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.9. CarbonDioxide function interface

The Yoctopuce class YCarbonDioxide allows you to read and configure Yoctopuce CO2 sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to perform manual calibration if reuired.

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

js	<script type='text/javascript' src='yocto_carbondioxide.js'></script>
cpp	#include "yocto_carbondioxide.h"
m	#import "yocto_carbondioxide.h"
pas	uses yocto_carbondioxide;
vb	yocto_carbondioxide.vb
cs	yocto_carbondioxide.cs
java	import com.yoctopuce.YoctoAPI.YCarbonDioxide;
uwp	import com.yoctopuce.YoctoAPI.YCarbonDioxide;
py	from yocto_carbondioxide import *
php	require_once('yocto_carbondioxide.php');
es	in HTML: <script src="../../lib/yocto_carbondioxide.js"></script> in node.js: require('yoctolib-es2017/yocto_carbondioxide.js');

Global functions
yFindCarbonDioxide(**func**) Retrieves a CO2 sensor for a given identifier.
yFindCarbonDioxideInContext(**yctx**, **func**) Retrieves a CO2 sensor for a given identifier in a YAPI context.
yFirstCarbonDioxide() Starts the enumeration of CO2 sensors currently accessible.
yFirstCarbonDioxideInContext(**yctx**) 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→clearCache() Invalidates the cache.
carbondioxide→describe() Returns a short text that describes unambiguously the instance of the CO2 sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
carbondioxide→get_abcPeriod() Returns the Automatic Baseline Calibration period, in hours.
carbondioxide→get_advertisedValue() Returns the current value of the CO2 sensor (no more than 6 characters).
carbondioxide→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.
carbondioxide→get_currentValue() Returns the current value of the CO2 concentration, in ppm (vol), as a floating point number.
carbondioxide→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.

carbondioxide→get_errorMessage()

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

carbondioxide→get_errorType()

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

carbondioxide→get_friendlyName()

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

carbondioxide→get_functionDescriptor()

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

carbondioxide→get_functionId()

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

carbondioxide→get_hardwareId()

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

carbondioxide→get_highestValue()

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

carbondioxide→get_logFrequency()

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

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

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

carbondioxide→get_module()

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

carbondioxide→get_module_async(**callback**, **context**)

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

carbondioxide→get_recordedData(**startTime**, **endTime**)

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

carbondioxide→get_reportFrequency()

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

carbondioxide→get_unit()

Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

carbondioxide→isOnline_async(**callback**, **context**)

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

carbondioxide→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

carbondioxide→load(**msValidity**)

Preloads the CO2 sensor cache with a specified validity duration.

carbondioxide→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

carbondioxide→loadCalibrationPoints(rawValues, refValues)

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

carbondioxide→load_async(msValidity, callback, context)

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

carbondioxide→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

carbondioxide→nextCarbonDioxide()

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

carbondioxide→registerTimedReportCallback(callback)

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

carbondioxide→registerValueCallback(callback)

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

carbondioxide→set_abcPeriod(newval)

Modifies Automatic Baseline Calibration period, in hours.

carbondioxide→set_highestValue(newval)

Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(newval)

Changes the logical name of the CO2 sensor.

carbondioxide→set_lowestValue(newval)

Changes the recorded minimal value observed.

carbondioxide→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

carbondioxide→set_resolution(newval)

Changes the resolution of the measured physical values.

carbondioxide→set_userData(data)

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

carbondioxide→startDataLogger()

Starts the data logger on the device.

carbondioxide→stopDataLogger()

Stops the datalogger on the device.

carbondioxide→triggerBaselineCalibration()

Triggers a baseline calibration at standard CO2 ambiant level (400ppm).

carbondioxide→triggerZeroCalibration()

Triggers a zero calibration of the sensor on carbon dioxide-free air.

carbondioxide→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

carbondioxide→wait_async(callback, context)

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

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

Retrieves a CO2 sensor for a given identifier.

function **FindCarbonDioxide**(**func**)

The identifier can be specified using several formats:

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

This function does not require that the CO2 sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCarbonDioxide.isOnline()` to test if the CO2 sensor is indeed online at a given time. In case of ambiguity when looking for a CO2 sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the CO2 sensor

Returns :

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

YCarbonDioxide.FindCarbonDioxideInContext() YCarbonDioxide yFindCarbonDioxideInContext()

Retrieves a CO2 sensor for a given identifier in a YAPI context.

function **FindCarbonDioxideInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the CO2 sensor

Returns :

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

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

Starts the enumeration of CO2 sensors currently accessible.

function **FirstCarbonDioxide**( )

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

Returns :

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

YCarbonDioxide.FirstCarbonDioxideInContext() YCarbonDioxide yFirstCarbonDioxideInContext()

Starts the enumeration of CO2 sensors currently accessible.

function **FirstCarbonDioxideInContext**(**yctx**)

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

Parameters :

yctx a YAPI context.

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() YCarbonDioxide
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.

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→clearCache() YCarbonDioxide
carbondioxide.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the CO2 sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

carbondioxide→describe()carbondioxide.describe() YCarbonDioxide

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

function **describe**( )

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

Returns :

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

carbondioxide→get_abcPeriod() YCarbonDioxide
carbondioxide→abcPeriod()
carbondioxide.get_abcPeriod()

Returns the Automatic Baseline Calibration period, in hours.

function **get_abcPeriod**( )

A negative value means that automatic baseline calibration is disabled.

Returns :
an integer corresponding to the Automatic Baseline Calibration period, in hours

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

carbondioxide→get_advertisedValue() YCarbonDioxide
carbondioxide→advertisedValue()
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() YCarbonDioxide
carbondioxide→currentRawValue()
carbondioxide.get_currentRawValue()

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

function **get_currentRawValue**( )

Returns :

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

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

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

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

function **get_currentValue**( )

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

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

carbondioxide→get_dataLogger() YCarbonDioxide
carbondioxide→dataLogger()
carbondioxide.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

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

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

function **get_errorMessage**( )

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

Returns :
a string corresponding to the latest error message that occured while using the CO2 sensor object

carbondioxide→get_errorType() YCarbonDioxide
carbondioxide→errorType()
carbondioxide.get_errorType()

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

function **get_errorType**( )

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

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

carbondioxide→get_friendlyName() YCarbonDioxide
carbondioxide→friendlyName()
carbondioxide.get_friendlyName()

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

function **get_friendlyName**( )

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

Returns :

a string that uniquely identifies the CO2 sensor using logical names (ex: `MyCustomName.relay1`)

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

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

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

function **get_functionDescriptor**( )

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

carbondioxide→get_functionId() YCarbonDioxide
carbondioxide→functionId()
carbondioxide.get_functionId()

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

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the CO2 sensor (ex: `relay1`)

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

carbondioxide→get_hardwareId() YCarbonDioxide
carbondioxide→hardwareId()
carbondioxide.get_hardwareId()

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

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the CO2 sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the CO2 sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

carbondioxide→get_highestValue() YCarbonDioxide
carbondioxide→highestValue()
carbondioxide.get_highestValue()

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

function **get_highestValue**( )

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

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

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

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

function **get_logFrequency**( )

Returns :

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

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

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

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

function **get_lowestValue**( )

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

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

carbondioxide→get_module() YCarbonDioxide
carbondioxide→module()
carbondioxide.get_module()

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

function **get_module**( )

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

Returns :
an instance of YModule

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

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

function **get_recordedData**(**startTime**, **endTime**)

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

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

Parameters :

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

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

Returns :

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

carbondioxide→get_reportFrequency() YCarbonDioxide
carbondioxide→reportFrequency()
carbondioxide.get_reportFrequency()

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

function **get_reportFrequency**( )

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

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

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

Returns the resolution of the measured values.

function **get_resolution**( )

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

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

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

carbondioxide→get_sensorState() YCarbonDioxide
carbondioxide→sensorState()
carbondioxide.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :
an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

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

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

Returns the measuring unit for the CO2 concentration.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the CO2 concentration

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

carbondioxide→get_userData() YCarbonDioxide
carbondioxide→userData()
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()carbondioxide.isOnline() YCarbonDioxide

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

function **isOnline**( )

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

Returns :

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

carbondioxide→load()carbondioxide.load() YCarbonDioxide

Preloads the CO2 sensor cache with a specified validity duration.

function **load**(**msValidity**)

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

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

carbondioxide→loadCalibrationPoints() YCarbonDioxide
carbondioxide.loadCalibrationPoints()

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

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→muteValueCallbacks() YCarbonDioxide
carbondioxide.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

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

carbondioxide→nextCarbonDioxide() YCarbonDioxide
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→registerTimedReportCallback() YCarbonDioxide
carbondioxide.registerTimedReportCallback()

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

function **registerTimedReportCallback**(**callback**)

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

Parameters :

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

carbondioxide→registerValueCallback() YCarbonDioxide
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_abcPeriod() YCarbonDioxide
carbondioxide→setAbcPeriod()
carbondioxide.set_abcPeriod()

Modifies Automatic Baseline Calibration period, in hours.

function **set_abcPeriod**(**newval**)

If you need to disable automatic baseline calibration (for instance when using the sensor in an environment that is constantly above 400ppm CO2), set the period to -1. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→set_highestValue() YCarbonDioxide
carbondioxide→setHighestValue()
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_logFrequency() YCarbonDioxide
carbondioxide→setLogFrequency()
carbondioxide.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→set_logicalName() YCarbonDioxide
carbondioxide→setLogicalName()
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() YCarbonDioxide
carbondioxide→setLowestValue()
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_reportFrequency() YCarbonDioxide
carbondioxide→setReportFrequency()
carbondioxide.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_resolution() YCarbonDioxide
carbondioxide→setResolution()
carbondioxide.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→set_userData() YCarbonDioxide
carbondioxide→setUserData()
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

carbondioxide→startDataLogger() YCarbonDioxide
carbondioxide.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

carbondioxide→stopDataLogger() YCarbonDioxide
carbondioxide.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

carbondioxide→triggerBaselineCalibration() YCarbonDioxide
carbondioxide.triggerBaselineCalibration()

Triggers a baseline calibration at standard CO2 ambiant level (400ppm).

function **triggerBaselineCalibration**( )

It is normally not necessary to manually calibrate the sensor, because the built-in automatic baseline calibration procedure will automatically fix any long-term drift based on the lowest level of CO2 observed over the automatic calibration period. However, if you disable automatic baseline calibration, you may want to manually trigger a calibration from time to time. Before starting a baseline calibration, make sure to put the sensor in a standard environment (e.g. outside in fresh air) at around 400ppm.

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→triggerZeroCalibration() YCarbonDioxide
carbondioxide.triggerZeroCalibration()

Triggers a zero calibration of the sensor on carbon dioxide-free air.

function **triggerZeroCalibration**( )

It is normally not necessary to manually calibrate the sensor, because the built-in automatic baseline calibration procedure will automatically fix any long-term drift based on the lowest level of CO2 observed over the automatic calibration period. However, if you disable automatic baseline calibration, you may want to manually trigger a calibration from time to time. Before starting a zero calibration, you should circulate carbon dioxide-free air within the sensor for a minute or two, using a small pipe connected to the sensor. Please contact support@yoctopuce.com for more details on the zero calibration procedure.

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→unmuteValueCallbacks() YCarbonDioxide
carbondioxide.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

carbondioxide→wait_async() YCarbonDioxide
carbondioxide.wait_async()

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

function **wait_async**(**callback**, **context**)

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

Parameters :

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

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

Returns :

nothing.

3.10. Cellular function interface

YCellular functions provides control over cellular network parameters and status for devices that are GSM-enabled.

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

js	<script type='text/javascript' src='yocto_cellular.js'></script>
cpp	#include "yocto_cellular.h"
m	#import "yocto_cellular.h"
pas	uses yocto_cellular;
vb	yocto_cellular.vb
cs	yocto_cellular.cs
java	import com.yoctopuce.YoctoAPI.YCellular;
uwp	import com.yoctopuce.YoctoAPI.YCellular;
py	from yocto_cellular import *
php	require_once('yocto_cellular.php');
es	in HTML: <script src="../../lib/yocto_cellular.js"></script> in node.js: require('yoctolib-es2017/yocto_cellular.js');

Global functions

yFindCellular(**func**)

Retrieves a cellular interface for a given identifier.

yFindCellularInContext(**yctx**, **func**)

Retrieves a cellular interface for a given identifier in a YAPI context.

yFirstCellular()

Starts the enumeration of cellular interfaces currently accessible.

yFirstCellularInContext(**yctx**)

Starts the enumeration of cellular interfaces currently accessible.

y_AT(**cmd**)

Sends an AT command to the GSM module and returns the command output.

`YCellular` methods

cellular→clearCache()

Invalidates the cache.

cellular→clearDataCounters()

Clear the transmitted data counters.

cellular→describe()

Returns a short text that describes unambiguously the instance of the cellular interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

cellular→get_advertisedValue()

Returns the current value of the cellular interface (no more than 6 characters).

cellular→get_airplaneMode()

Returns true if the airplane mode is active (radio turned off).

cellular→get_apn()

Returns the Access Point Name (APN) to be used, if needed.

cellular→get_apnSecret()

Returns an opaque string if APN authentication parameters have been configured in the device, or an empty string otherwise.

cellular→get_availableOperators()

Returns the list detected cell operators in the neighborhood.

cellular→get_cellIdentifier()

Returns the unique identifier of the cellular antenna in use: MCC, MNC, LAC and Cell ID.

cellular→get_cellOperator()

Returns the name of the cell operator currently in use.

cellular→get_cellType()

Active cellular connection type.

cellular→get_dataReceived()

Returns the number of bytes received so far.

cellular→get_dataSent()

Returns the number of bytes sent so far.

cellular→get_enableData()

Returns the condition for enabling IP data services (GPRS).

cellular→get_errorMessage()

Returns the error message of the latest error with the cellular interface.

cellular→get_errorType()

Returns the numerical error code of the latest error with the cellular interface.

cellular→get_friendlyName()

Returns a global identifier of the cellular interface in the format `MODULE_NAME.FUNCTION_NAME`.

cellular→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

cellular→get_functionId()

Returns the hardware identifier of the cellular interface, without reference to the module.

cellular→get_hardwareId()

Returns the unique hardware identifier of the cellular interface in the form `SERIAL.FUNCTIONID`.

cellular→get_imsi()

Returns an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card.

cellular→get_linkQuality()

Returns the link quality, expressed in percent.

cellular→get_lockedOperator()

Returns the name of the only cell operator to use if automatic choice is disabled, or an empty string if the SIM card will automatically choose among available cell operators.

cellular→get_logicalName()

Returns the logical name of the cellular interface.

cellular→get_message()

Returns the latest status message from the wireless interface.

cellular→get_module()

Gets the `YModule` object for the device on which the function is located.

cellular→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

cellular→get_pin()

Returns an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card.

cellular→get_pingInterval()

Returns the automated connectivity check interval, in seconds.

cellular→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

cellular→isOnline()

Checks if the cellular interface is currently reachable, without raising any error.

cellular→isOnline_async(**callback**, **context**)

Checks if the cellular interface is currently reachable, without raising any error (asynchronous version).

cellular→load(**msValidity**)

Preloads the cellular interface cache with a specified validity duration.

cellular→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cellular→load_async(**msValidity**, **callback**, **context**)

Preloads the cellular interface cache with a specified validity duration (asynchronous version).

cellular→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cellular→nextCellular()

Continues the enumeration of cellular interfaces started using `yFirstCellular()`.

cellular→quickCellSurvey()

Returns a list of nearby cellular antennas, as required for quick geolocation of the device.

cellular→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

cellular→sendPUK(**puk**, **newPin**)

Sends a PUK code to unlock the SIM card after three failed PIN code attempts, and setup a new PIN into the SIM card.

cellular→set_airplaneMode(**newval**)

Changes the activation state of airplane mode (radio turned off).

cellular→set_apn(**newval**)

Returns the Access Point Name (APN) to be used, if needed.

cellular→set_apnAuth(**username**, **password**)

Configure authentication parameters to connect to the APN.

cellular→set_dataReceived(**newval**)

Changes the value of the incoming data counter.

cellular→set_dataSent(**newval**)

Changes the value of the outgoing data counter.

cellular→set_enableData(**newval**)

Changes the condition for enabling IP data services (GPRS).

cellular→set_lockedOperator(**newval**)

Changes the name of the cell operator to be used.

cellular→set_logicalName(**newval**)

Changes the logical name of the cellular interface.

cellular→set_pin(**newval**)

Changes the PIN code used by the module to access the SIM card.

cellular→set_pingInterval(**newval**)

Changes the automated connectivity check interval, in seconds.

cellular→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

cellular→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cellular→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YCellular.FindCellular() YCellular yFindCellular()yFindCellular()

Retrieves a cellular interface for a given identifier.

```
function FindCellular( 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 cellular interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCellular.isOnline()` to test if the cellular interface is indeed online at a given time. In case of ambiguity when looking for a cellular 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the cellular interface

Returns :

a `YCellular` object allowing you to drive the cellular interface.

YCellular.FindCellularInContext() YCellular
yFindCellularInContext()

Retrieves a cellular interface for a given identifier in a YAPI context.

function **FindCellularInContext**(**yctx**, **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 cellular interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCellular.isOnline()` to test if the cellular interface is indeed online at a given time. In case of ambiguity when looking for a cellular 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 :

yctx a YAPI context

func a string that uniquely characterizes the cellular interface

Returns :

a `YCellular` object allowing you to drive the cellular interface.

YCellular.FirstCellular() YCellular
yFirstCellular()yFirstCellular()

Starts the enumeration of cellular interfaces currently accessible.

function **FirstCellular**( )

Use the method `YCellular.nextCellular()` to iterate on next cellular interfaces.

Returns :

a pointer to a `YCellular` object, corresponding to the first cellular interface currently online, or a `null` pointer if there are none.

YCellular.FirstCellularInContext() YCellular
yFirstCellularInContext()

Starts the enumeration of cellular interfaces currently accessible.

function **FirstCellularInContext**(**yctx**)

Use the method `YCellular.nextCellular()` to iterate on next cellular interfaces.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YCellular` object, corresponding to the first cellular interface currently online, or a `null` pointer if there are none.

YCellular._AT() YCellular y_AT()

Sends an AT command to the GSM module and returns the command output.

```
function _AT( cmd)
```

The command will only execute when the GSM module is in standard command state, and should leave it in the exact same state. Use this function with great care !

Parameters :

cmd the AT command to execute, like for instance: "+CCLK?".

Returns :

a string with the result of the commands. Empty lines are automatically removed from the output.

cellular→clearCache()cellular.clearCache() YCellular

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the cellular interface attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

cellular→clearDataCounters() YCellular
cellular.clearDataCounters()

Clear the transmitted data counters.

function **clearDataCounters**( )

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→describe()cellular.describe() YCellular

Returns a short text that describes unambiguously the instance of the cellular interface in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the cellular interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

cellular→get_advertisedValue() YCellular
cellular→advertisedValue()
cellular.get_advertisedValue()

Returns the current value of the cellular interface (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the cellular interface (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

cellular→get_airplaneMode() YCellular
cellular→airplaneMode()cellular.get_airplaneMode()

Returns true if the airplane mode is active (radio turned off).

function **get_airplaneMode**( )

Returns :

either Y_AIRPLANEMODE_OFF or Y_AIRPLANEMODE_ON, according to true if the airplane mode is active (radio turned off)

On failure, throws an exception or returns Y_AIRPLANEMODE_INVALID.

cellular→get_apn() YCellular
cellular→apn()cellular.get_apn()

Returns the Access Point Name (APN) to be used, if needed.

```
function get_apn( )
```

When left blank, the APN suggested by the cell operator will be used.

Returns :
a string corresponding to the Access Point Name (APN) to be used, if needed

On failure, throws an exception or returns `Y_APN_INVALID`.

cellular→get_apnSecret() YCellular
cellular→apnSecret()cellular.get_apnSecret()

Returns an opaque string if APN authentication parameters have been configured in the device, or an empty string otherwise.

function **get_apnSecret**( )

To configure these parameters, use `set_apnAuth()`.

Returns :

a string corresponding to an opaque string if APN authentication parameters have been configured in the device, or an empty string otherwise

On failure, throws an exception or returns `Y_APNSECRET_INVALID`.

cellular→get_availableOperators() YCellular
cellular→availableOperators()
cellular.get_availableOperators()

Returns the list detected cell operators in the neighborhood.

function **get_availableOperators**( )

This function will typically take between 30 seconds to 1 minute to return. Note that any SIM card can usually only connect to specific operators. All networks returned by this function might therefore not be available for connection.

Returns :
a list of string (cell operator names).

cellular→get_cellIdentifier() YCellular
cellular→cellIdentifier()cellular.get_cellIdentifier()

Returns the unique identifier of the cellular antenna in use: MCC, MNC, LAC and Cell ID.

function **get_cellIdentifier**( )

Returns :
a string corresponding to the unique identifier of the cellular antenna in use: MCC, MNC, LAC and Cell ID

On failure, throws an exception or returns Y_CELLIDENTIFIER_INVALID.

cellular→get_cellOperator() YCellular
cellular→cellOperator()cellular.get_cellOperator()

Returns the name of the cell operator currently in use.

function **get_cellOperator**( )

Returns :
a string corresponding to the name of the cell operator currently in use

On failure, throws an exception or returns `Y_CELLOPERATOR_INVALID`.

cellular→get_cellType() YCellular
cellular→cellType()cellular.get_cellType()

Active cellular connection type.

function **get_cellType**( )

Returns :

a value among `Y_CELLTYPE_GPRS`, `Y_CELLTYPE_EGPRS`, `Y_CELLTYPE_WCDMA`, `Y_CELLTYPE_HSDPA`, `Y_CELLTYPE_NONE` and `Y_CELLTYPE_CDMA`

On failure, throws an exception or returns `Y_CELLTYPE_INVALID`.

cellular→get_dataReceived() YCellular
cellular→dataReceived()cellular.get_dataReceived()

Returns the number of bytes received so far.

function **get_dataReceived**( )

Returns :
an integer corresponding to the number of bytes received so far

On failure, throws an exception or returns `Y_DATARECEIVED_INVALID`.

cellular→get_dataSent() YCellular
cellular→dataSent()cellular.get_dataSent()

Returns the number of bytes sent so far.

function **get_dataSent**( )

Returns :
an integer corresponding to the number of bytes sent so far

On failure, throws an exception or returns `Y_DATASENT_INVALID`.

cellular→get_enableData() YCellular
cellular→enableData()cellular.get_enableData()

Returns the condition for enabling IP data services (GPRS).

function **get_enableData**( )

When data services are disabled, SMS are the only mean of communication.

Returns :

a value among Y_ENABLEDATA_HOMENETWORK, Y_ENABLEDATA_ROAMING, Y_ENABLEDATA_NEVER and Y_ENABLEDATA_NEUTRALITY corresponding to the condition for enabling IP data services (GPRS)

On failure, throws an exception or returns Y_ENABLEDATA_INVALID.

cellular→get_errorMessage() YCellular
cellular→errorMessage()cellular.get_errorMessage()

Returns the error message of the latest error with the cellular interface.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the cellular interface object

cellular→get_errorType() YCellular
cellular→errorType()cellular.get_errorType()

Returns the numerical error code of the latest error with the cellular interface.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the cellular interface object

cellular→get_friendlyName() YCellular
cellular→friendlyName()cellular.get_friendlyName()

Returns a global identifier of the cellular interface in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the cellular interface if they are defined, otherwise the serial number of the module and the hardware identifier of the cellular interface (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the cellular interface using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

cellular→get_functionDescriptor() YCellular
cellular→functionDescriptor()
cellular.get_functionDescriptor()

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.

cellular→get_functionId() YCellular
cellular→functionId()cellular.get_functionId()

Returns the hardware identifier of the cellular interface, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the cellular interface (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

cellular→get_hardwareId() YCellular
cellular→hardwareId()cellular.get_hardwareId()

Returns the unique hardware identifier of the cellular interface in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the cellular interface (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the cellular interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

cellular→get_imsi() YCellular
cellular→imsi()cellular.get_imsi()

Returns an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card.

function **get_imsi**( )

Returns :

a string corresponding to an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card

On failure, throws an exception or returns `Y_IMSI_INVALID`.

cellular→get_linkQuality() YCellular
cellular→linkQuality()cellular.get_linkQuality()

Returns the link quality, expressed in percent.

function **get_linkQuality**( )

Returns :
an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns `Y_LINKQUALITY_INVALID`.

cellular→get_lockedOperator() YCellular
cellular→lockedOperator()
cellular.get_lockedOperator()

Returns the name of the only cell operator to use if automatic choice is disabled, or an empty string if the SIM card will automatically choose among available cell operators.

function **get_lockedOperator**( )

Returns :

a string corresponding to the name of the only cell operator to use if automatic choice is disabled, or an empty string if the SIM card will automatically choose among available cell operators

On failure, throws an exception or returns `Y_LOCKEDOPERATOR_INVALID`.

cellular→get_logicalName() YCellular
cellular→logicalName()cellular.get_logicalName()

Returns the logical name of the cellular interface.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the cellular interface.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

cellular→get_message() YCellular
cellular→message()cellular.get_message()

Returns the latest status message from the wireless interface.

function **get_message**( )

Returns :
a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns `Y_MESSAGE_INVALID`.

cellular→get_module() YCellular
cellular→module()cellular.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

cellular→get_pin() YCellular
cellular→pin()cellular.get_pin()

Returns an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card.

function **get_pin**( )

Returns :

a string corresponding to an opaque string if a PIN code has been configured in the device to access the SIM card, or an empty string if none has been configured or if the code provided was rejected by the SIM card

On failure, throws an exception or returns `Y_PIN_INVALID`.

cellular→get_pingInterval() YCellular
cellular→pingInterval()cellular.get_pingInterval()

Returns the automated connectivity check interval, in seconds.

function **get_pingInterval**( )

Returns :
an integer corresponding to the automated connectivity check interval, in seconds

On failure, throws an exception or returns `Y_PINGINTERVAL_INVALID`.

cellular→get_userData() YCellular
cellular→userData()cellular.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.

cellular→isOnline()cellular.isOnline() YCellular

Checks if the cellular interface is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the cellular interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the cellular interface.

Returns :

`true` if the cellular interface can be reached, and `false` otherwise

cellular→load()cellular.load() YCellular

Preloads the cellular interface cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

cellular→loadAttribute()cellular.loadAttribute() YCellular

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

cellular→muteValueCallbacks() YCellular
cellular.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→nextCellular()cellular.nextCellular() YCellular

Continues the enumeration of cellular interfaces started using `yFirstCellular()`.

function **nextCellular**( )

Returns :

a pointer to a `YCellular` object, corresponding to a cellular interface currently online, or a `null` pointer if there are no more cellular interfaces to enumerate.

cellular→quickCellSurvey()cellular.quickCellSurvey() YCellular

Returns a list of nearby cellular antennas, as required for quick geolocation of the device.

function **quickCellSurvey**( )

The first cell listed is the serving cell, and the next ones are the neighboor cells reported by the serving cell.

Returns :
a list of YCellRecords.

cellular→registerValueCallback() YCellular
cellular.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.

cellular→sendPUK()cellular.sendPUK() YCellular

Sends a PUK code to unlock the SIM card after three failed PIN code attempts, and setup a new PIN into the SIM card.

function **sendPUK**(**puk**, **newPin**)

Only ten consecutives tentatives are permitted: after that, the SIM card will be blocked permanently without any mean of recovery to use it again. Note that after calling this method, you have usually to invoke method `set_pin()` to tell the YoctoHub which PIN to use in the future.

Parameters :

puk the SIM PUK code

newPin new PIN code to configure into the SIM card

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_airplaneMode() YCellular
cellular→setAirplaneMode()
cellular.set_airplaneMode()

Changes the activation state of airplane mode (radio turned off).

function **set_airplaneMode**(**newval**)

Parameters :

newval either `Y_AIRPLANEMODE_OFF` or `Y_AIRPLANEMODE_ON`, according to the activation state of airplane mode (radio turned off)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_apn() YCellular
cellular→setApn()cellular.set_apn()

Returns the Access Point Name (APN) to be used, if needed.

function **set_apn**(**newval**)

When left blank, the APN suggested by the cell operator will be used.

Parameters :

newval a string

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_apnAuth() YCellular
cellular→setApnAuth()cellular.set_apnAuth()

Configure authentication parameters to connect to the APN.

function **set_apnAuth**(**username**, **password**)

Both PAP and CHAP authentication are supported.

Parameters :

username APN username

password APN password

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_dataReceived() YCellular
cellular→setDataReceived()
cellular.set_dataReceived()

Changes the value of the incoming data counter.

function **set_dataReceived**(**newval**)

Parameters :

newval an integer corresponding to the value of the incoming data counter

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_dataSent() YCellular
cellular→setDataSent()cellular.set_dataSent()

Changes the value of the outgoing data counter.

function **set_dataSent**(**newval**)

Parameters :

newval an integer corresponding to the value of the outgoing data counter

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_enableData() YCellular
cellular→setEnableData()cellular.set_enableData()

Changes the condition for enabling IP data services (GPRS).

function **set_enableData**(**newval**)

The service can be either fully deactivated, or limited to the SIM home network, or enabled for all partner networks (roaming). Caution: enabling data services on roaming networks may cause prohibitive communication costs !

When data services are disabled, SMS are the only mean of communication.

Parameters :

newval a value among `Y_ENABLEDATA_HOMENETWORK`, `Y_ENABLEDATA_ROAMING`, `Y_ENABLEDATA_NEVER` and `Y_ENABLEDATA_NEUTRALITY` corresponding to the condition for enabling IP data services (GPRS)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_lockedOperator() YCellular
cellular→setLockedOperator()
cellular.set_lockedOperator()

Changes the name of the cell operator to be used.

function **set_lockedOperator**(**newval**)

If the name is an empty string, the choice will be made automatically based on the SIM card. Otherwise, the selected operator is the only one that will be used.

Parameters :

newval a string corresponding to the name of the cell operator to be used

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_logicalName() YCellular
cellular→setLogicalName()cellular.set_logicalName()

Changes the logical name of the cellular 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 cellular interface.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_pin() YCellular
cellular→setPin()cellular.set_pin()

Changes the PIN code used by the module to access the SIM card.

function **set_pin**(**newval**)

This function does not change the code on the SIM card itself, but only changes the parameter used by the device to try to get access to it. If the SIM code does not work immediately on first try, it will be automatically forgotten and the message will be set to "Enter SIM PIN". The method should then be invoked again with right correct PIN code. After three failed attempts in a row, the message is changed to "Enter SIM PUK" and the SIM card PUK code must be provided using method `sendPUK`.

Remember to call the `saveToFlash()` method of the module to save the new value in the device flash.

Parameters :

newval a string corresponding to the PIN code used by the module to access the SIM card

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_pingInterval() YCellular
cellular→setPingInterval()cellular.set_pingInterval()

Changes the automated connectivity check interval, in seconds.

function **set_pingInterval**(**newval**)

Parameters :

newval an integer corresponding to the automated connectivity check interval, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→set_userData() YCellular
cellular→setUserData()cellular.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

cellular→unmuteValueCallbacks() YCellular
cellular.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

cellular→wait_async()cellular.wait_async() YCellular

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.11. ColorLed function interface

The Yoctopuce application programming interface allows you to drive a color LED using RGB coordinates as well as HSL coordinates. The module performs all conversions form RGB to HSL automatically. It is then self-evident to turn on a LED with a given hue and to progressively vary its saturation or lightness. If needed, you can find more information on the difference between RGB and HSL in the section following this one.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_colorled.js'></script>
cpp     #include "yocto_colorled.h"
m       #import "yocto_colorled.h"
pas     uses yocto_colorled;
vb      yocto_colorled.vb
cs      yocto_colorled.cs
java    import com.yoctopuce.YoctoAPI.YColorLed;
uwp     import com.yoctopuce.YoctoAPI.YColorLed;
py      from yocto_colorled import *
php     require_once('yocto_colorled.php');
es      in HTML: <script src="../../lib/yocto_colorled.js"></script>
        in node.js: require('yoctolib-es2017/yocto_colorled.js');
```

Global functions
yFindColorLed(**func**) Retrieves an RGB LED for a given identifier.
yFindColorLedInContext(**yctx**, **func**) Retrieves an RGB LED for a given identifier in a YAPI context.
yFirstColorLed() Starts the enumeration of RGB LEDs currently accessible.
yFirstColorLedInContext(**yctx**) Starts the enumeration of RGB LEDs currently accessible.
`YColorLed` methods
colorled→addHslMoveToBlinkSeq(**HSLcolor**, **msDelay**) Add a new transition to the blinking sequence, the move will be performed in the HSL space.
colorled→addRgbMoveToBlinkSeq(**RGBcolor**, **msDelay**) Adds a new transition to the blinking sequence, the move is performed in the RGB space.
colorled→clearCache() Invalidates the cache.
colorled→describe() Returns a short text that describes unambiguously the instance of the RGB LED in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
colorled→get_advertisedValue() Returns the current value of the RGB LED (no more than 6 characters).
colorled→get_blinkSeqMaxSize() Returns the maximum length of the blinking sequence.
colorled→get_blinkSeqSignature() Return the blinking sequence signature.
colorled→get_blinkSeqSize()

Returns the current length of the blinking sequence.

colorled→get_errorMessage()
Returns the error message of the latest error with the RGB LED.

colorled→get_errorType()
Returns the numerical error code of the latest error with the RGB LED.

colorled→get_friendlyName()
Returns a global identifier of the RGB LED in the format `MODULE_NAME.FUNCTION_NAME`.

colorled→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

colorled→get_functionId()
Returns the hardware identifier of the RGB LED, without reference to the module.

colorled→get_hardwareId()
Returns the unique hardware identifier of the RGB LED in the form `SERIAL.FUNCTIONID`.

colorled→get_hslColor()
Returns the current HSL color of the LED.

colorled→get_logicalName()
Returns the logical name of the RGB LED.

colorled→get_module()
Gets the `YModule` object for the device on which the function is located.

colorled→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

colorled→get_rgbColor()
Returns the current RGB color of the LED.

colorled→get_rgbColorAtPowerOn()
Returns the configured color to be displayed when the module is turned on.

colorled→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

colorled→hslMove(**hsl_target**, **ms_duration**)
Performs a smooth transition in the HSL color space between the current color and a target color.

colorled→isOnline()
Checks if the RGB LED is currently reachable, without raising any error.

colorled→isOnline_async(**callback**, **context**)
Checks if the RGB LED is currently reachable, without raising any error (asynchronous version).

colorled→load(**msValidity**)
Preloads the RGB LED cache with a specified validity duration.

colorled→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

colorled→load_async(**msValidity**, **callback**, **context**)
Preloads the RGB LED cache with a specified validity duration (asynchronous version).

colorled→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

colorled→nextColorLed()
Continues the enumeration of RGB LEDs started using `yFirstColorLed()`.

colorled→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

colorled→resetBlinkSeq()

Resets the preprogrammed blinking sequence.

colorled→rgbMove(**rgb_target**, **ms_duration**)

Performs a smooth transition in the RGB color space between the current color and a target color.

colorled→set_hslColor(**newval**)

Changes the current color of the LED, using a color HSL.

colorled→set_logicalName(**newval**)

Changes the logical name of the RGB LED.

colorled→set_rgbColor(**newval**)

Changes the current color of the LED, using an RGB color.

colorled→set_rgbColorAtPowerOn(**newval**)

Changes the color that the LED will display by default when the module is turned on.

colorled→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

colorled→startBlinkSeq()

Starts the preprogrammed blinking sequence.

colorled→stopBlinkSeq()

Stops the preprogrammed blinking sequence.

colorled→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

colorled→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YColorLed.FindColorLed() YColorLed yFindColorLed()yFindColorLed()

Retrieves an RGB LED for a given identifier.

```
function FindColorLed( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the RGB LED is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YColorLed.isOnline()` to test if the RGB LED is indeed online at a given time. In case of ambiguity when looking for an RGB LED by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the RGB LED

Returns :

a `YColorLed` object allowing you to drive the RGB LED.

YColorLed.FindColorLedInContext() YColorLed yFindColorLedInContext()

Retrieves an RGB LED for a given identifier in a YAPI context.

function **FindColorLedInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the RGB LED

Returns :

a `YColorLed` object allowing you to drive the RGB LED.

YColorLed.FirstColorLed() YColorLed yFirstColorLed()yFirstColorLed()

Starts the enumeration of RGB LEDs currently accessible.

function **FirstColorLed**( )

Use the method `YColorLed.nextColorLed()` to iterate on next RGB LEDs.

Returns :

a pointer to a `YColorLed` object, corresponding to the first RGB LED currently online, or a `null` pointer if there are none.

YColorLed.FirstColorLedInContext() yFirstColorLedInContext()

YColorLed

Starts the enumeration of RGB LEDs currently accessible.

function **FirstColorLedInContext**(**yctx**)

Use the method `YColorLed.nextColorLed()` to iterate on next RGB LEDs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YColorLed` object, corresponding to the first RGB LED currently online, or a `null` pointer if there are none.

colorled→addHslMoveToBlinkSeq() YColorLed
colorled.addHslMoveToBlinkSeq()

Add a new transition to the blinking sequence, the move will be performed in the HSL space.

function **addHslMoveToBlinkSeq**(**HSLcolor**, **msDelay**)

Parameters :

HSLcolor desired HSL color when the traisntion is completed

msDelay duration of the color transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

colorled→addRgbMoveToBlinkSeq() YColorLed
colorled.addRgbMoveToBlinkSeq()

Adds a new transition to the blinking sequence, the move is performed in the RGB space.

function **addRgbMoveToBlinkSeq**(**RGBcolor**, **msDelay**)

Parameters :

RGBcolor desired RGB color when the transition is completed

msDelay duration of the color transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

colorled→clearCache()colorled.clearCache() YColorLed

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the RGB LED attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

colorled→describe()colorled.describe() YColorLed

Returns a short text that describes unambiguously the instance of the RGB LED in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the RGB LED (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

colorled→get_advertisedValue() YColorLed
colorled→advertisedValue()
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_blinkSeqMaxSize() YColorLed
colorled→blinkSeqMaxSize()
colorled.get_blinkSeqMaxSize()

Returns the maximum length of the blinking sequence.

function **get_blinkSeqMaxSize**( )

Returns :
an integer corresponding to the maximum length of the blinking sequence

On failure, throws an exception or returns `Y_BLINKSEQMAXSIZE_INVALID`.

colorled→get_blinkSeqSignature() YColorLed
colorled→blinkSeqSignature()
colorled.get_blinkSeqSignature()

Return the blinking sequence signature.

function **get_blinkSeqSignature**( )

Since blinking sequences cannot be read from the device, this can be used to detect if a specific blinking sequence is already programmed.

Returns :
an integer

On failure, throws an exception or returns `Y_BLINKSEQSIGNATURE_INVALID`.

colorled→get_blinkSeqSize() YColorLed
colorled→blinkSeqSize()colorled.get_blinkSeqSize()

Returns the current length of the blinking sequence.

function **get_blinkSeqSize**( )

Returns :
an integer corresponding to the current length of the blinking sequence

On failure, throws an exception or returns `Y_BLINKSEQSIZE_INVALID`.

colorled→get_errorMessage() YColorLed
colorled→errorMessage()
colorled.get_errorMessage()

Returns the error message of the latest error with the RGB LED.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the RGB LED object

colorled→get_errorType() YColorLed
colorled→errorType()colorled.get_errorType()

Returns the numerical error code of the latest error with the RGB LED.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the RGB LED object

colorled→get_friendlyName() YColorLed
colorled→friendlyName()colorled.get_friendlyName()

Returns a global identifier of the RGB LED in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the RGB LED if they are defined, otherwise the serial number of the module and the hardware identifier of the RGB LED (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the RGB LED using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

colorled→get_functionDescriptor() YColorLed
colorled→functionDescriptor()
colorled.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

colorled→get_functionId() YColorLed
colorled→functionId()colorled.get_functionId()

Returns the hardware identifier of the RGB LED, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the RGB LED (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

colorled→get_hardwareId() YColorLed
colorled→hardwareId()colorled.get_hardwareId()

Returns the unique hardware identifier of the RGB LED in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the RGB LED (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the RGB LED (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

colorled→get_hslColor() YColorLed
colorled→hslColor()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() YColorLed
colorled→logicalName()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() YColorLed
colorled→module()colorled.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

colorled→get_rgbColor() YColorLed
colorled→rgbColor()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() YColorLed
colorled→rgbColorAtPowerOn()
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() YColorLed
colorled→userData()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→hslMove()colorled.hslMove() YColorLed

Performs a smooth transition in the HSL color space between the current color and a target color.

function **hslMove**(**hsl_target**, **ms_duration**)

Parameters :

hsl_target	desired HSL color at the end of the transition
ms_duration	duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorled→isOnline()colorled.isOnline() YColorLed

Checks if the RGB LED is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the RGB LED in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the RGB LED.

Returns :
`true` if the RGB LED can be reached, and `false` otherwise

colorled→load()colorled.load() YColorLed

Preloads the RGB LED cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()colorled.loadAttribute() YColorLed

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

colorled→muteValueCallbacks() YColorLed
colorled.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorled→nextColorLed()colorled.nextColorLed() YColorLed

Continues the enumeration of RGB LEDs started using `yFirstColorLed()`.

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() YColorLed
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→resetBlinkSeq()colorled.resetBlinkSeq() YColorLed

Resets the preprogrammed blinking sequence.

function **resetBlinkSeq**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

colorled→rgbMove()colorled.rgbMove() YColorLed

Performs a smooth transition in the RGB color space between the current color and a target color.

function **rgbMove**(**rgb_target**, **ms_duration**)

Parameters :

rgb_target desired RGB color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorled→set_hslColor() YColorLed
colorled→setHslColor()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() YColorLed
colorled→setLogicalName()
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() YColorLed
colorled→setRgbColor()colorled.set_rgbColor()

Changes the current color of the LED, using an 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 an RGB color

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorled→set_rgbColorAtPowerOn() YColorLed
colorled→setRgbColorAtPowerOn()
colorled.set_rgbColorAtPowerOn()

Changes the color that the LED will display by default when the module is turned on.

function **set_rgbColorAtPowerOn**(**newval**)

Parameters :

newval an integer corresponding to the color that the LED will display by default when the module is turned on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorled→set_userData() YColorLed
colorled→setUserData()colorled.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

function **set_userData**(**data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

colorled→startBlinkSeq()colorled.startBlinkSeq() YColorLed

Starts the preprogrammed blinking sequence.

function **startBlinkSeq**( )

The sequence is run in a loop until it is stopped by stopBlinkSeq or an explicit change.

Returns :
YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

colorled→stopBlinkSeq()colorled.stopBlinkSeq() YColorLed

Stops the preprogrammed blinking sequence.

function **stopBlinkSeq**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

colorled→unmuteValueCallbacks() YColorLed
colorled.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorled→wait_async()colorled.wait_async() YColorLed

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.12. ColorLedCluster function interface

The Yoctopuce application programming interface allows you to drive a color LED cluster. Unlike the ColorLed class, the ColorLedCluster allows to handle several LEDs at one. Color changes can be done using RGB coordinates as well as HSL coordinates. The module performs all conversions form RGB to HSL automatically. It is then self-evident to turn on a LED with a given hue and to progressively vary its saturation or lightness. If needed, you can find more information on the difference between RGB and HSL in the section following this one.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_colorledcluster.js'></script>
cpp	#include "yocto_colorledcluster.h"
m	#import "yocto_colorledcluster.h"
pas	uses yocto_colorledcluster;
vb	yocto_colorledcluster.vb
cs	yocto_colorledcluster.cs
java	import com.yoctopuce.YoctoAPI.YColorLedCluster;
uwp	import com.yoctopuce.YoctoAPI.YColorLedCluster;
py	from yocto_colorledcluster import *
php	require_once('yocto_colorledcluster.php');
es	in HTML: <script src="../../lib/yocto_colorledcluster.js"></script> in node.js: require('yoctolib-es2017/yocto_colorledcluster.js');

Global functions

yFindColorLedCluster(**func**)

Retrieves a RGB LED cluster for a given identifier.

yFindColorLedClusterInContext(**yctx**, **func**)

Retrieves a RGB LED cluster for a given identifier in a YAPI context.

yFirstColorLedCluster()

Starts the enumeration of RGB LED clusters currently accessible.

yFirstColorLedClusterInContext(**yctx**)

Starts the enumeration of RGB LED clusters currently accessible.

`YColorLedCluster` methods

colorledcluster→addHslMoveToBlinkSeq(**seqIndex**, **hslValue**, **delay**)

Adds an HSL transition to a sequence.

colorledcluster→addMirrorToBlinkSeq(**seqIndex**)

Adds a mirror ending to a sequence.

colorledcluster→addRgbMoveToBlinkSeq(**seqIndex**, **rgbValue**, **delay**)

Adds an RGB transition to a sequence.

colorledcluster→clearCache()

Invalidates the cache.

colorledcluster→describe()

Returns a short text that describes unambiguously the instance of the RGB LED cluster in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

colorledcluster→get_activeLedCount()

Returns the number of LEDs currently handled by the device.

colorledcluster→get_advertisedValue()

Returns the current value of the RGB LED cluster (no more than 6 characters).

colorledcluster→get_blinkSeqMaxCount()

Returns the maximum number of sequences that the device can handle.

colorledcluster→get_blinkSeqMaxSize()
Returns the maximum length of sequences.

colorledcluster→get_blinkSeqSignatures(**seqIndex**, **count**)
Returns a list on 32 bit signatures for specified blinking sequences.

colorledcluster→get_blinkSeqState(**seqIndex**, **count**)
Returns a list of integers with the started state for specified blinking sequences.

colorledcluster→get_blinkSeqStateAtPowerOn(**seqIndex**, **count**)
Returns a list of integers with the "auto-start at power on" flag state for specified blinking sequences.

colorledcluster→get_blinkSeqStateSpeed(**seqIndex**, **count**)
Returns a list of integers with the current speed for specified blinking sequences.

colorledcluster→get_errorMessage()
Returns the error message of the latest error with the RGB LED cluster.

colorledcluster→get_errorType()
Returns the numerical error code of the latest error with the RGB LED cluster.

colorledcluster→get_friendlyName()
Returns a global identifier of the RGB LED cluster in the format `MODULE_NAME.FUNCTION_NAME`.

colorledcluster→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

colorledcluster→get_functionId()
Returns the hardware identifier of the RGB LED cluster, without reference to the module.

colorledcluster→get_hardwareId()
Returns the unique hardware identifier of the RGB LED cluster in the form `SERIAL.FUNCTIONID`.

colorledcluster→get_linkedSeqArray(**ledIndex**, **count**)
Returns a list on sequence index for each RGB LED.

colorledcluster→get_logicalName()
Returns the logical name of the RGB LED cluster.

colorledcluster→get_maxLedCount()
Returns the maximum number of LEDs that the device can handle.

colorledcluster→get_module()
Gets the `YModule` object for the device on which the function is located.

colorledcluster→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

colorledcluster→get_rgbColorArray(**ledIndex**, **count**)
Returns a list on 24bit RGB color values with the current colors displayed on the RGB leds.

colorledcluster→get_rgbColorArrayAtPowerOn(**ledIndex**, **count**)
Returns a list on 24bit RGB color values with the RGB LEDs startup colors.

colorledcluster→get_rgbColorBuffer(**ledIndex**, **count**)
Returns a binary buffer with content from the LED RGB buffer, as is.

colorledcluster→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

colorledcluster→hslArray_move(**hslList**, **delay**)
Sets up a smooth HSL color transition to the specified pixel-by-pixel list of HSL color codes.

colorledcluster→hsl_move(**ledIndex**, **count**, **hslValue**, **delay**)

Allows you to modify the current color of a group of adjacent LEDs to another color, in a seamless and autonomous manner.

colorledcluster→isOnline()

Checks if the RGB LED cluster is currently reachable, without raising any error.

colorledcluster→isOnline_async(**callback**, **context**)

Checks if the RGB LED cluster is currently reachable, without raising any error (asynchronous version).

colorledcluster→linkLedToBlinkSeq(**ledIndex**, **count**, **seqIndex**, **offset**)

Links adjacent LEDs to a specific sequence.

colorledcluster→linkLedToBlinkSeqAtPowerOn(**ledIndex**, **count**, **seqIndex**, **offset**)

Links adjacent LEDs to a specific sequence at device poweron.

colorledcluster→linkLedToPeriodicBlinkSeq(**ledIndex**, **count**, **seqIndex**, **periods**)

Links adjacent LEDs to a specific sequence.

colorledcluster→load(**msValidity**)

Preloads the RGB LED cluster cache with a specified validity duration.

colorledcluster→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

colorledcluster→load_async(**msValidity**, **callback**, **context**)

Preloads the RGB LED cluster cache with a specified validity duration (asynchronous version).

colorledcluster→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

colorledcluster→nextColorLedCluster()

Continues the enumeration of RGB LED clusters started using `yFirstColorLedCluster()`.

colorledcluster→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

colorledcluster→resetBlinkSeq(**seqIndex**)

Stops a sequence execution and resets its contents.

colorledcluster→rgbArray_move(**rgbList**, **delay**)

Sets up a smooth RGB color transition to the specified pixel-by-pixel list of RGB color codes.

colorledcluster→rgb_move(**ledIndex**, **count**, **rgbValue**, **delay**)

Allows you to modify the current color of a group of adjacent LEDs to another color, in a seamless and autonomous manner.

colorledcluster→saveBlinkSeq(**seqIndex**)

Saves the definition of a sequence.

colorledcluster→saveLedsConfigAtPowerOn()

Saves the LEDs power-on configuration.

colorledcluster→set_activeLedCount(**newval**)

Changes the number of LEDs currently handled by the device.

colorledcluster→set_blinkSeqSpeed(**seqIndex**, **speed**)

Changes the execution speed of a sequence.

colorledcluster→set_blinkSeqStateAtPowerOn(**seqIndex**, **autostart**)

Configures a sequence to make it start automatically at device startup.

colorledcluster→set_hslColor(**ledIndex**, **count**, **hslValue**)

Changes the current color of consecutive LEDs in the cluster, using a HSL color.

colorledcluster→set_hslColorArray(**ledIndex**, **hslList**)

Sends 24bit HSL colors (provided as a list of integers) to the LED HSL buffer, as is.

colorledcluster→set_hslColorBuffer(**ledIndex**, **buff**)

Sends a binary buffer to the LED HSL buffer, as is.

colorledcluster→set_logicalName(newval)

Changes the logical name of the RGB LED cluster.

colorledcluster→set_rgbColor(ledIndex, count, rgbValue)

Changes the current color of consecutve LEDs in the cluster, using a RGB color.

colorledcluster→set_rgbColorArray(ledIndex, rgbList)

Sends 24bit RGB colors (provided as a list of integers) to the LED RGB buffer, as is.

colorledcluster→set_rgbColorAtPowerOn(ledIndex, count, rgbValue)

Changes the color at device startup of consecutve LEDs in the cluster, using a RGB color.

colorledcluster→set_rgbColorBuffer(ledIndex, buff)

Sends a binary buffer to the LED RGB buffer, as is.

colorledcluster→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

colorledcluster→startBlinkSeq(seqIndex)

Starts a sequence execution: every LED linked to that sequence starts to run it in a loop.

colorledcluster→stopBlinkSeq(seqIndex)

Stops a sequence execution.

colorledcluster→unlinkLedFromBlinkSeq(ledIndex, count)

Unlinks adjacent LEDs from a sequence.

colorledcluster→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

colorledcluster→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YColorLedCluster.FindColorLedCluster() YColorLedCluster yFindColorLedCluster()yFindColorLedCluster()

Retrieves a RGB LED cluster for a given identifier.

```
function FindColorLedCluster( 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 cluster is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YColorLedCluster.isOnline()` to test if the RGB LED cluster is indeed online at a given time. In case of ambiguity when looking for a RGB LED cluster 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the RGB LED cluster

Returns :

a `YColorLedCluster` object allowing you to drive the RGB LED cluster.

YColorLedCluster.FindColorLedClusterInContext() YColorLedCluster yFindColorLedClusterInContext()

Retrieves a RGB LED cluster for a given identifier in a YAPI context.

function **FindColorLedClusterInContext**(**yctx**, **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 cluster is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YColorLedCluster.isOnline()` to test if the RGB LED cluster is indeed online at a given time. In case of ambiguity when looking for a RGB LED cluster 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 :

yctx a YAPI context

func a string that uniquely characterizes the RGB LED cluster

Returns :

a `YColorLedCluster` object allowing you to drive the RGB LED cluster.

YColorLedCluster.FirstColorLedCluster() YColorLedCluster yFirstColorLedCluster()yFirstColorLedCluster()

Starts the enumeration of RGB LED clusters currently accessible.

function **FirstColorLedCluster**( )

Use the method `YColorLedCluster.nextColorLedCluster()` to iterate on next RGB LED clusters.

Returns :

a pointer to a `YColorLedCluster` object, corresponding to the first RGB LED cluster currently online, or a `null` pointer if there are none.

YColorLedCluster.FirstColorLedClusterInContext() YColorLedCluster yFirstColorLedClusterInContext()

Starts the enumeration of RGB LED clusters currently accessible.

function **FirstColorLedClusterInContext**(**yctx**)

Use the method `YColorLedCluster.nextColorLedCluster()` to iterate on next RGB LED clusters.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YColorLedCluster` object, corresponding to the first RGB LED cluster currently online, or a `null` pointer if there are none.

colorledcluster→addHslMoveToBlinkSeq() YColorLedCluster
colorledcluster.addHslMoveToBlinkSeq()

Adds an HSL transition to a sequence.

function **addHslMoveToBlinkSeq**(**seqIndex**, **hslValue**, **delay**)

A sequence is a transition list, which can be executed in loop by an group of LEDs. Sequences are persistant and are saved in the device flash memory as soon as the `saveBlinkSeq()` method is called.

Parameters :

seqIndex	sequence index.
hslValue	target color (0xHHSSLL)
delay	transition duration in ms

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→addMirrorToBlinkSeq() YColorLedCluster
colorledcluster.addMirrorToBlinkSeq()

Adds a mirror ending to a sequence.

function **addMirrorToBlinkSeq**(**seqIndex**)

When the sequence will reach the end of the last transition, its running speed will automatically be reversed so that the sequence plays in the reverse direction, like in a mirror. After the first transition of the sequence is played at the end of the reverse execution, the sequence starts again in the initial direction.

Parameters :

seqIndex sequence index.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→addRgbMoveToBlinkSeq() YColorLedCluster
colorledcluster.addRgbMoveToBlinkSeq()

Adds an RGB transition to a sequence.

function **addRgbMoveToBlinkSeq**(**seqIndex**, **rgbValue**, **delay**)

A sequence is a transition list, which can be executed in loop by a group of LEDs. Sequences are persistent and are saved in the device flash memory as soon as the `saveBlinkSeq()` method is called.

Parameters :

seqIndex	sequence index.
rgbValue	target color (0xRRGGBB)
delay	transition duration in ms

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→clearCache() YColorLedCluster
colorledcluster.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the RGB LED cluster attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

colorledcluster→describe()colorledcluster.describe() YColorLedCluster

Returns a short text that describes unambiguously the instance of the RGB LED cluster in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the RGB LED cluster (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

colorledcluster→get_activeLedCount() YColorLedCluster
colorledcluster→activeLedCount()
colorledcluster.get_activeLedCount()

Returns the number of LEDs currently handled by the device.

function **get_activeLedCount**( )

Returns :
an integer corresponding to the number of LEDs currently handled by the device

On failure, throws an exception or returns `Y_ACTIVELEDCOUNT_INVALID`.

colorledcluster→get_advertisedValue() YColorLedCluster
colorledcluster→advertisedValue()
colorledcluster.get_advertisedValue()

Returns the current value of the RGB LED cluster (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the RGB LED cluster (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

colorledcluster→get_blinkSeqMaxCount() YColorLedCluster
colorledcluster→blinkSeqMaxCount()
colorledcluster.get_blinkSeqMaxCount()

Returns the maximum number of sequences that the device can handle.

function **get_blinkSeqMaxCount**( )

Returns :
an integer corresponding to the maximum number of sequences that the device can handle

On failure, throws an exception or returns `Y_BLINKSEQMAXCOUNT_INVALID`.

colorledcluster→get_blinkSeqMaxSize() YColorLedCluster
colorledcluster→blinkSeqMaxSize()
colorledcluster.get_blinkSeqMaxSize()

Returns the maximum length of sequences.

function **get_blinkSeqMaxSize**( )

Returns :
an integer corresponding to the maximum length of sequences

On failure, throws an exception or returns `Y_BLINKSEQMAXSIZE_INVALID`.

colorledcluster→get_blinkSeqSignatures() YColorLedCluster
colorledcluster→blinkSeqSignatures()
colorledcluster.get_blinkSeqSignatures()

Returns a list on 32 bit signatures for specified blinking sequences.

function **get_blinkSeqSignatures**(**seqIndex**, **count**)

Since blinking sequences cannot be read from the device, this can be used to detect if a specific blinking sequence is already programmed.

Parameters :

seqIndex index of the first blinking sequence which should be returned

count number of blinking sequences which should be returned

Returns :

a list of 32 bit integer signatures

On failure, throws an exception or returns an empty array.

colorledcluster→get_blinkSeqState() YColorLedCluster
colorledcluster→blinkSeqState()
colorledcluster.get_blinkSeqState()

Returns a list of integers with the started state for specified blinking sequences.

function **get_blinkSeqState**(**seqIndex**, **count**)

Parameters :

seqIndex index of the first blinking sequence which should be returned

count number of blinking sequences which should be returned

Returns :

a list of integers, 0 for sequences turned off and 1 for sequences running

On failure, throws an exception or returns an empty array.

colorledcluster→get_blinkSeqStateAtPowerOn() YColorLedCluster
colorledcluster→blinkSeqStateAtPowerOn()
colorledcluster.get_blinkSeqStateAtPowerOn()

Returns a list of integers with the "auto-start at power on" flag state for specified blinking sequences.

function **get_blinkSeqStateAtPowerOn**(**seqIndex**, **count**)

Parameters :

seqIndex index of the first blinking sequence which should be returned

count number of blinking sequences which should be returned

Returns :

a list of integers, 0 for sequences turned off and 1 for sequences running

On failure, throws an exception or returns an empty array.

colorledcluster→get_blinkSeqStateSpeed() YColorLedCluster
colorledcluster→blinkSeqStateSpeed()
colorledcluster.get_blinkSeqStateSpeed()

Returns a list of integers with the current speed for specified blinking sequences.

function **get_blinkSeqStateSpeed**(**seqIndex**, **count**)

Parameters :

seqIndex index of the first sequence speed which should be returned

count number of sequence speeds which should be returned

Returns :

a list of integers, 0 for sequences turned off and 1 for sequences running

On failure, throws an exception or returns an empty array.

colorledcluster→get_errorMessage() YColorLedCluster
colorledcluster→errorMessage()
colorledcluster.get_errorMessage()

Returns the error message of the latest error with the RGB LED cluster.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the RGB LED cluster object

colorledcluster→get_errorType() YColorLedCluster
colorledcluster→errorType()
colorledcluster.get_errorType()

Returns the numerical error code of the latest error with the RGB LED cluster.

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 the RGB LED cluster object

colorledcluster→get_friendlyName() YColorLedCluster
colorledcluster→friendlyName()
colorledcluster.get_friendlyName()

Returns a global identifier of the RGB LED cluster in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the RGB LED cluster if they are defined, otherwise the serial number of the module and the hardware identifier of the RGB LED cluster (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the RGB LED cluster using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

colorledcluster→get_functionDescriptor() YColorLedCluster
colorledcluster→functionDescriptor()
colorledcluster.get_functionDescriptor()

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.

colorledcluster→get_functionId() YColorLedCluster
colorledcluster→functionId()
colorledcluster.get_functionId()

Returns the hardware identifier of the RGB LED cluster, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the RGB LED cluster (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

colorledcluster→get_hardwareId() YColorLedCluster
colorledcluster→hardwareId()
colorledcluster.get_hardwareId()

Returns the unique hardware identifier of the RGB LED cluster in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the RGB LED cluster (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the RGB LED cluster (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

colorledcluster→get_linkedSeqArray() YColorLedCluster
colorledcluster→linkedSeqArray()
colorledcluster.get_linkedSeqArray()

Returns a list on sequence index for each RGB LED.

function **get_linkedSeqArray**(**ledIndex**, **count**)

The first number represents the sequence index for the the first LED, the second number represents the sequence index for the second LED, etc.

Parameters :

ledIndex index of the first LED which should be returned

count number of LEDs which should be returned

Returns :

a list of integers with sequence index

On failure, throws an exception or returns an empty array.

colorledcluster→get_logicalName() YColorLedCluster
colorledcluster→logicalName()
colorledcluster.get_logicalName()

Returns the logical name of the RGB LED cluster.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the RGB LED cluster.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

colorledcluster→get_maxLedCount() YColorLedCluster
colorledcluster→maxLedCount()
colorledcluster.get_maxLedCount()

Returns the maximum number of LEDs that the device can handle.

function **get_maxLedCount**( )

Returns :
an integer corresponding to the maximum number of LEDs that the device can handle

On failure, throws an exception or returns `Y_MAXLEDCOUNT_INVALID`.

colorledcluster→get_module() YColorLedCluster
colorledcluster→module()
colorledcluster.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

colorledcluster→get_rgbColorArray() YColorLedCluster
colorledcluster→rgbColorArray()
colorledcluster.get_rgbColorArray()

Returns a list on 24bit RGB color values with the current colors displayed on the RGB leds.

function **get_rgbColorArray**(**ledIndex**, **count**)

The first number represents the RGB value of the first LED, the second number represents the RGB value of the second LED, etc.

Parameters :

ledIndex index of the first LED which should be returned

count number of LEDs which should be returned

Returns :

a list of 24bit color codes with RGB components of selected LEDs, as 0xRRGGBB.

On failure, throws an exception or returns an empty array.

colorledcluster→get_rgbColorArrayAtPowerOn() YColorLedCluster
colorledcluster→rgbColorArrayAtPowerOn()
colorledcluster.get_rgbColorArrayAtPowerOn()

Returns a list on 24bit RGB color values with the RGB LEDs startup colors.

function **get_rgbColorArrayAtPowerOn**(**ledIndex**, **count**)

The first number represents the startup RGB value of the first LED, the second number represents the RGB value of the second LED, etc.

Parameters :

ledIndex index of the first LED which should be returned

count number of LEDs which should be returned

Returns :

a list of 24bit color codes with RGB components of selected LEDs, as 0xRRGGBB.

On failure, throws an exception or returns an empty array.

colorledcluster→get_rgbColorBuffer() YColorLedCluster
colorledcluster→rgbColorBuffer()
colorledcluster.get_rgbColorBuffer()

Returns a binary buffer with content from the LED RGB buffer, as is.

function **get_rgbColorBuffer**(**ledIndex**, **count**)

First three bytes are RGB components for the first LED in the interval, the next three bytes for the second LED in the interval, etc.

Parameters :

ledIndex index of the first LED which should be returned

count number of LEDs which should be returned

Returns :

a binary buffer with RGB components of selected LEDs.

On failure, throws an exception or returns an empty binary buffer.

colorledcluster→get_userData() YColorLedCluster
colorledcluster→userData()
colorledcluster.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.

colorledcluster→hslArray_move() YColorLedCluster
colorledcluster.hslArray_move()

Sets up a smooth HSL color transition to the specified pixel-by-pixel list of HSL color codes.

function **hslArray_move**(**hslList**, **delay**)

The first color code represents the target HSL value of the first LED, the second color code represents the target value of the second LED, etc.

Parameters :

hslList a list of target 24bit HSL codes, in the form 0xHHSSLL

delay transition duration in ms

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→hsl_move() YColorLedCluster
colorledcluster.hsl_move()

Allows you to modify the current color of a group of adjacent LEDs to another color, in a seamless and autonomous manner.

function **hsl_move**(**ledIndex**, **count**, **hslValue**, **delay**)

The transition is performed in the HSL space. In HSL, hue is a circular value (0..360°). There are always two paths to perform the transition: by increasing or by decreasing the hue. The module selects the shortest transition. If the difference is exactly 180°, the module selects the transition which increases the hue.

Parameters :

ledIndex	index of the fisrt affected LED.
count	affected LED count.
hslValue	new color (0xHHSSLL).
delay	transition duration in ms

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→isOnline()colorledcluster.isOnline() YColorLedCluster

Checks if the RGB LED cluster is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the RGB LED cluster in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the RGB LED cluster.

Returns :
`true` if the RGB LED cluster can be reached, and `false` otherwise

colorledcluster→linkLedToBlinkSeq() YColorLedCluster
colorledcluster.linkLedToBlinkSeq()

Links adjacent LEDs to a specific sequence.

function **linkLedToBlinkSeq**(**ledIndex**, **count**, **seqIndex**, **offset**)

These LEDs start to execute the sequence as soon as startBlinkSeq is called. It is possible to add an offset in the execution: that way we can have several groups of LED executing the same sequence, with a temporal offset. A LED cannot be linked to more than one sequence.

Parameters :

ledIndex	index of the first affected LED.
count	affected LED count.
seqIndex	sequence index.
offset	execution offset in ms.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→linkLedToBlinkSeqAtPowerOn() YColorLedCluster
colorledcluster.linkLedToBlinkSeqAtPowerOn()

Links adjacent LEDs to a specific sequence at device poweron.

function **linkLedToBlinkSeqAtPowerOn**(**ledIndex**, **count**, **seqIndex**, **offset**)

Don't forget to configure the sequence auto start flag as well and call `saveLedsConfigAtPowerOn()`. It is possible to add an offset in the execution: that way we can have several groups of LEDs executing the same sequence, with a temporal offset. A LED cannot be linked to more than one sequence.

Parameters :

ledIndex	index of the first affected LED.
count	affected LED count.
seqIndex	sequence index.
offset	execution offset in ms.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→linkLedToPeriodicBlinkSeq() YColorLedCluster
colorledcluster.linkLedToPeriodicBlinkSeq()

Links adjacent LEDs to a specific sequence.

function **linkLedToPeriodicBlinkSeq**(**ledIndex**, **count**, **seqIndex**, **periods**)

These LED start to execute the sequence as soon as startBlinkSeq is called. This function automatically introduces a shift between LEDs so that the specified number of sequence periods appears on the group of LEDs (wave effect).

Parameters :

ledIndex	index of the first affected LED.
count	affected LED count.
seqIndex	sequence index.
periods	number of periods to show on LEDs.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→load()colorledcluster.load() YColorLedCluster

Preloads the RGB LED cluster 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.

colorledcluster→loadAttribute() YColorLedCluster
colorledcluster.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

colorledcluster→muteValueCallbacks() YColorLedCluster
colorledcluster.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→nextColorLedCluster() YColorLedCluster
colorledcluster.nextColorLedCluster()

Continues the enumeration of RGB LED clusters started using `yFirstColorLedCluster()`.

function **nextColorLedCluster**( )

Returns :

a pointer to a `YColorLedCluster` object, corresponding to a RGB LED cluster currently online, or a `null` pointer if there are no more RGB LED clusters to enumerate.

colorledcluster→registerValueCallback() YColorLedCluster
colorledcluster.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.

colorledcluster→resetBlinkSeq() YColorLedCluster
colorledcluster.resetBlinkSeq()

Stops a sequence execution and resets its contents.

function **resetBlinkSeq**(**seqIndex**)

Leds linked to this sequence are not automatically updated anymore.

Parameters :

seqIndex index of the sequence to reset

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→rgbArray_move() YColorLedCluster
colorledcluster.rgbArray_move()

Sets up a smooth RGB color transition to the specified pixel-by-pixel list of RGB color codes.

function **rgbArray_move**(**rgbList**, **delay**)

The first color code represents the target RGB value of the first LED, the next color code represents the target value of the next LED, etc.

Parameters :

rgbList a list of target 24bit RGB codes, in the form 0xRRGGBB

delay transition duration in ms

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→rgb_move() YColorLedCluster
colorledcluster.rgb_move()

Allows you to modify the current color of a group of adjacent LEDs to another color, in a seamless and autonomous manner.

function **rgb_move**(**ledIndex**, **count**, **rgbValue**, **delay**)

The transition is performed in the RGB space.

Parameters :

ledIndex	index of the first affected LED.
count	affected LED count.
rgbValue	new color (0xRRGGBB).
delay	transition duration in ms

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→saveBlinkSeq() YColorLedCluster
colorledcluster.saveBlinkSeq()

Saves the definition of a sequence.

function **saveBlinkSeq**(**seqIndex**)

Warning: only sequence steps and flags are saved. to save the LEDs startup bindings, the method `saveLedsConfigAtPowerOn()` must be called.

Parameters :

seqIndex index of the sequence to start.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→saveLedsConfigAtPowerOn() YColorLedCluster
colorledcluster.saveLedsConfigAtPowerOn()

Saves the LEDs power-on configuration.

function **saveLedsConfigAtPowerOn**( )

This includes the start-up color or sequence binding for all LEDs. Warning: if some LEDs are linked to a sequence, the method `saveBlinkSeq()` must also be called to save the sequence definition.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_activeLedCount() YColorLedCluster
colorledcluster→setActiveLedCount()
colorledcluster.set_activeLedCount()

Changes the number of LEDs currently handled by the device.

function **set_activeLedCount**(**newval**)

Parameters :

newval an integer corresponding to the number of LEDs currently handled by the device

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_blinkSeqSpeed() YColorLedCluster
colorledcluster→setBlinkSeqSpeed()
colorledcluster.set_blinkSeqSpeed()

Changes the execution speed of a sequence.

function **set_blinkSeqSpeed**(**seqIndex**, **speed**)

The natural execution speed is 1000 per thousand. If you configure a slower speed, you can play the sequence in slow-motion. If you set a negative speed, you can play the sequence in reverse direction.

Parameters :

seqIndex index of the sequence to start.

speed sequence running speed (-1000...1000).

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_blinkSeqStateAtPowerOn() YColorLedCluster
colorledcluster→setBlinkSeqStateAtPowerOn()
colorledcluster.set_blinkSeqStateAtPowerOn()

Configures a sequence to make it start automatically at device startup.

function **set_blinkSeqStateAtPowerOn**(**seqIndex**, **autostart**)

Don't forget to call `saveBlinkSeq()` to make sure the modification is saved in the device flash memory.

Parameters :

seqIndex index of the sequence to reset.

autostart 0 to keep the sequence turned off and 1 to start it automatically.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_hslColor() YColorLedCluster
colorledcluster→setHslColor()
colorledcluster.set_hslColor()

Changes the current color of consecutive LEDs in the cluster, using a HSL color.

function **set_hslColor**(**ledIndex**, **count**, **hslValue**)

Encoding is done as follows: 0xHHSSLL.

Parameters :

ledIndex index of the first affected LED.

count affected LED count.

hslValue new color.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_hslColorArray() YColorLedCluster
colorledcluster→setHslColorArray()
colorledcluster.set_hslColorArray()

Sends 24bit HSL colors (provided as a list of integers) to the LED HSL buffer, as is.

function **set_hslColorArray**(**ledIndex**, **hslList**)

The first number represents the HSL value of the LED specified as parameter, the second number represents the HSL value of the second LED, etc.

Parameters :

ledIndex index of the first LED which should be updated

hslList a list of 24bit HSL codes, in the form 0xHHSSLL

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_hslColorBuffer() YColorLedCluster
colorledcluster→setHslColorBuffer()
colorledcluster.set_hslColorBuffer()

Sends a binary buffer to the LED HSL buffer, as is.

function **set_hslColorBuffer**(**ledIndex**, **buff**)

First three bytes are HSL components for the LED specified as parameter, the next three bytes for the second LED, etc.

Parameters :

ledIndex index of the first LED which should be updated

buff the binary buffer to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_logicalName() YColorLedCluster
colorledcluster→setLogicalName()
colorledcluster.set_logicalName()

Changes the logical name of the RGB LED cluster.

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 cluster.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_rgbColor() YColorLedCluster
colorledcluster→setRgbColor()
colorledcluster.set_rgbColor()

Changes the current color of consecutve LEDs in the cluster, using a RGB color.

function **set_rgbColor**(**ledIndex**, **count**, **rgbValue**)

Encoding is done as follows: 0xRRGGBB.

Parameters :

ledIndex index of the first affected LED.
count affected LED count.
rgbValue new color.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_rgbColorArray() YColorLedCluster
colorledcluster→setRgbColorArray()
colorledcluster.set_rgbColorArray()

Sends 24bit RGB colors (provided as a list of integers) to the LED RGB buffer, as is.

function **set_rgbColorArray**(**ledIndex**, **rgbList**)

The first number represents the RGB value of the LED specified as parameter, the second number represents the RGB value of the next LED, etc.

Parameters :

ledIndex index of the first LED which should be updated

rgbList a list of 24bit RGB codes, in the form 0xRRGGBB

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_rgbColorAtPowerOn() YColorLedCluster
colorledcluster→setRgbColorAtPowerOn()
colorledcluster.set_rgbColorAtPowerOn()

Changes the color at device startup of consecutve LEDs in the cluster, using a RGB color.

function **set_rgbColorAtPowerOn**(**ledIndex**, **count**, **rgbValue**)

Encoding is done as follows: 0xRRGGBB. Don't forget to call `saveLedsConfigAtPowerOn()` to make sure the modification is saved in the device flash memory.

Parameters :

ledIndex index of the first affected LED.

count affected LED count.

rgbValue new color.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_rgbColorBuffer() YColorLedCluster
colorledcluster→setRgbColorBuffer()
colorledcluster.set_rgbColorBuffer()

Sends a binary buffer to the LED RGB buffer, as is.

function **set_rgbColorBuffer**(**ledIndex**, **buff**)

First three bytes are RGB components for LED specified as parameter, the next three bytes for the next LED, etc.

Parameters :

ledIndex index of the first LED which should be updated

buff the binary buffer to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→set_userData() YColorLedCluster
colorledcluster→setUserData()
colorledcluster.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

colorledcluster→startBlinkSeq() YColorLedCluster
colorledcluster.startBlinkSeq()

Starts a sequence execution: every LED linked to that sequence starts to run it in a loop.

function **startBlinkSeq**(**seqIndex**)

Parameters :

seqIndex index of the sequence to start.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→stopBlinkSeq() YColorLedCluster
colorledcluster.stopBlinkSeq()

Stops a sequence execution.

```
function stopBlinkSeq( seqIndex)
```

If started again, the execution restarts from the beginning.

Parameters :

seqIndex index of the sequence to stop.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→unlinkLedFromBlinkSeq() YColorLedCluster
colorledcluster.unlinkLedFromBlinkSeq()

Unlinks adjacent LEDs from a sequence.

function **unlinkLedFromBlinkSeq**(**ledIndex**, **count**)

Parameters :

ledIndex index of the first affected LED.

count affected LED count.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→unmuteValueCallbacks() YColorLedCluster
colorledcluster.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

colorledcluster→wait_async() YColorLedCluster
colorledcluster.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.13. Compass function interface

The YSensor class is the parent class for all Yoctopuce sensors. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provide a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButton class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_compass.js'></script>
cpp     #include "yocto_compass.h"
m       #import "yocto_compass.h"
pas     uses yocto_compass;
vb      yocto_compass.vb
cs      yocto_compass.cs
java    import com.yoctopuce.YoctoAPI.YCompass;
uwp     import com.yoctopuce.YoctoAPI.YCompass;
py      from yocto_compass import *
php     require_once('yocto_compass.php');
es      in HTML: <script src="../../lib/yocto_compass.js"></script>
        in node.js: require('yoctolib-es2017/yocto_compass.js');
```

Global functions
yFindCompass(**func**) Retrieves a compass for a given identifier.
yFindCompassInContext(**yctx**, **func**) Retrieves a compass for a given identifier in a YAPI context.
yFirstCompass() Starts the enumeration of compasses currently accessible.
yFirstCompassInContext(**yctx**) Starts the enumeration of compasses currently accessible.
YCompass methods
compass→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
compass→clearCache() Invalidates the cache.
compass→describe() Returns a short text that describes unambiguously the instance of the compass in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
compass→get_advertisedValue() Returns the current value of the compass (no more than 6 characters).
compass→get_bandwidth() Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).
compass→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.
compass→get_currentValue() Returns the current value of the relative bearing, in degrees, as a floating point number.

compass→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

compass→get_errorMessage()

Returns the error message of the latest error with the compass.

compass→get_errorType()

Returns the numerical error code of the latest error with the compass.

compass→get_friendlyName()

Returns a global identifier of the compass in the format `MODULE_NAME.FUNCTION_NAME`.

compass→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

compass→get_functionId()

Returns the hardware identifier of the compass, without reference to the module.

compass→get_hardwareId()

Returns the unique hardware identifier of the compass in the form `SERIAL.FUNCTIONID`.

compass→get_highestValue()

Returns the maximal value observed for the relative bearing since the device was started.

compass→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

compass→get_logicalName()

Returns the logical name of the compass.

compass→get_lowestValue()

Returns the minimal value observed for the relative bearing since the device was started.

compass→get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

compass→get_module()

Gets the `YModule` object for the device on which the function is located.

compass→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

compass→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

compass→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

compass→get_resolution()

Returns the resolution of the measured values.

compass→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

compass→get_unit()

Returns the measuring unit for the relative bearing.

compass→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

compass→isOnline()

Checks if the compass is currently reachable, without raising any error.

compass→isOnline_async(**callback**, **context**)

Checks if the compass is currently reachable, without raising any error (asynchronous version).

compass→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

compass→load(**msValidity**)

Preloads the compass cache with a specified validity duration.

compass→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

compass→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

compass→load_async(**msValidity**, **callback**, **context**)

Preloads the compass cache with a specified validity duration (asynchronous version).

compass→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

compass→nextCompass()

Continues the enumeration of compasses started using `yFirstCompass()`.

compass→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

compass→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

compass→set_bandwidth(**newval**)

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

compass→set_highestValue(**newval**)

Changes the recorded maximal value observed.

compass→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

compass→set_logicalName(**newval**)

Changes the logical name of the compass.

compass→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

compass→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

compass→set_resolution(**newval**)

Changes the resolution of the measured physical values.

compass→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

compass→startDataLogger()

Starts the data logger on the device.

compass→stopDataLogger()

Stops the datalogger on the device.

compass→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

compass→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YCompass.FindCompass() YCompass
yFindCompass()yFindCompass()

Retrieves a compass for a given identifier.

```
function FindCompass( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the compass is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCompass.isOnline()` to test if the compass is indeed online at a given time. In case of ambiguity when looking for a compass by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the compass

Returns :

a `YCompass` object allowing you to drive the compass.

YCompass.FindCompassInContext() YCompass yFindCompassInContext()

Retrieves a compass for a given identifier in a YAPI context.

function **FindCompassInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the compass is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCompass.isOnline()` to test if the compass is indeed online at a given time. In case of ambiguity when looking for a compass by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the compass

Returns :

a `YCompass` object allowing you to drive the compass.

YCompass.FirstCompass() YCompass yFirstCompass()yFirstCompass()

Starts the enumeration of compasses currently accessible.

function **FirstCompass**( )

Use the method `YCompass.nextCompass()` to iterate on next compasses.

Returns :

a pointer to a `YCompass` object, corresponding to the first compass currently online, or a `null` pointer if there are none.

YCompass.FirstCompassInContext() YCompass yFirstCompassInContext()

Starts the enumeration of compasses currently accessible.

function **FirstCompassInContext**(**yctx**)

Use the method `YCompass.nextCompass()` to iterate on next compasses.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YCompass` object, corresponding to the first compass currently online, or a `null` pointer if there are none.

compass→calibrateFromPoints() YCompass
compass.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→clearCache()compass.clearCache() YCompass

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the compass attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

compass→describe()compass.describe() YCompass

Returns a short text that describes unambiguously the instance of the compass in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the compass (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

compass→get_advertisedValue() YCompass
compass→advertisedValue()
compass.get_advertisedValue()

Returns the current value of the compass (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the compass (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

compass→get_bandwidth() YCompass
compass→bandwidth()compass.get_bandwidth()

Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **get_bandwidth**( )

Returns :

an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

On failure, throws an exception or returns `Y_BANDWIDTH_INVALID`.

compass→get_currentRawValue() YCompass
compass→currentRawValue()
compass.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

compass→get_currentValue() YCompass
compass→currentValue()
compass.get_currentValue()

Returns the current value of the relative bearing, in degrees, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the relative bearing, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

compass→get_dataLogger() YCompass
compass→dataLogger()compass.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

compass→get_errorMessage() YCompass
compass→errorMessage()
compass.get_errorMessage()

Returns the error message of the latest error with the compass.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the compass object

compass→get_errorType() YCompass
compass→errorType()compass.get_errorType()

Returns the numerical error code of the latest error with the compass.

```
function get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the compass object

compass→get_friendlyName() YCompass
compass→friendlyName()
compass.get_friendlyName()

Returns a global identifier of the compass in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the compass if they are defined, otherwise the serial number of the module and the hardware identifier of the compass (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the compass using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

compass→get_functionDescriptor() YCompass
compass→functionDescriptor()
compass.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

compass→get_functionId() YCompass
compass→functionId()compass.get_functionId()

Returns the hardware identifier of the compass, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the compass (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

compass→get_hardwareId() YCompass
compass→hardwareId()compass.get_hardwareId()

Returns the unique hardware identifier of the compass in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the compass (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the compass (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

compass→get_highestValue() YCompass
compass→highestValue()
compass.get_highestValue()

Returns the maximal value observed for the relative bearing since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the relative bearing since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

compass→get_logFrequency() YCompass
compass→logFrequency()
compass.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

compass→get_logicalName() YCompass
compass→logicalName()compass.get_logicalName()

Returns the logical name of the compass.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the compass.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

compass→get_lowestValue() YCompass
compass→lowestValue()compass.get_lowestValue()

Returns the minimal value observed for the relative bearing since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the relative bearing since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

compass→get_magneticHeading() YCompass
compass→magneticHeading()
compass.get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

function **get_magneticHeading**( )

Returns :
a floating point number corresponding to the magnetic heading, regardless of the configured bearing

On failure, throws an exception or returns `Y_MAGNETICHEADING_INVALID`.

compass→get_module() YCompass
compass→module()compass.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

compass→get_recordedData() YCompass
compass→recordedData()
compass.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

compass→get_reportFrequency() YCompass
compass→reportFrequency()
compass.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

compass→get_resolution() YCompass
compass→resolution()compass.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

compass→get_sensorState() YCompass
compass→sensorState()compass.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

compass→get_unit() YCompass
compass→unit()compass.get_unit()

Returns the measuring unit for the relative bearing.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the relative bearing

On failure, throws an exception or returns `Y_UNIT_INVALID`.

compass→get_userData() YCompass
compass→userData()compass.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.

compass→isOnline()compass.isOnline() YCompass

Checks if the compass is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the compass in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the compass.

Returns :

`true` if the compass can be reached, and `false` otherwise

compass→load()compass.load() YCompass

Preloads the compass cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→loadAttribute()compass.loadAttribute() YCompass

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

compass→loadCalibrationPoints() YCompass
compass.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→muteValueCallbacks() YCompass
compass.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→nextCompass()compass.nextCompass() YCompass

Continues the enumeration of compasses started using `yFirstCompass()`.

function **nextCompass**( )

Returns :

a pointer to a `YCompass` object, corresponding to a compass currently online, or a `null` pointer if there are no more compasses to enumerate.

compass→registerTimedReportCallback() YCompass
compass.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

compass→registerValueCallback() YCompass
compass.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.

compass→set_bandwidth() YCompass
compass→setBandwidth()compass.set_bandwidth()

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **set_bandwidth**(**newval**)

When the frequency is lower, the device performs averaging.

Parameters :

newval an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→set_highestValue() YCompass
compass→setHighestValue()
compass.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.

compass→set_logFrequency() YCompass
compass→setLogFrequency()
compass.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→set_logicalName() YCompass
compass→setLogicalName()
compass.set_logicalName()

Changes the logical name of the compass.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the compass.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→set_lowestValue() YCompass
compass→setLowestValue()
compass.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.

compass→set_reportFrequency() YCompass
compass→setReportFrequency()
compass.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→set_resolution() YCompass
compass→setResolution()compass.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→set_userData() YCompass
compass→setUserData()compass.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

compass→startDataLogger() YCompass
compass.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

compass→stopDataLogger() YCompass
compass.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

compass→unmuteValueCallbacks() YCompass
compass.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

compass→wait_async()compass.wait_async() YCompass

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.14. Current function interface

The Yoctopuce class YCurrent allows you to read and configure Yoctopuce current sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_current.js'></script>
cpp     #include "yocto_current.h"
m       #import "yocto_current.h"
pas     uses yocto_current;
vb      yocto_current.vb
cs      yocto_current.cs
java    import com.yoctopuce.YoctoAPI.YCurrent;
uwp     import com.yoctopuce.YoctoAPI.YCurrent;
py      from yocto_current import *
php     require_once('yocto_current.php');
es      in HTML: <script src="../../lib/yocto_current.js"></script>
        in node.js: require('yoctolib-es2017/yocto_current.js');
```

Global functions
yFindCurrent(**func**) Retrieves a current sensor for a given identifier.
yFindCurrentInContext(**yctx**, **func**) Retrieves a current sensor for a given identifier in a YAPI context.
yFirstCurrent() Starts the enumeration of current sensors currently accessible.
yFirstCurrentInContext(**yctx**) 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→clearCache() Invalidates the cache.
current→describe() Returns a short text that describes unambiguously the instance of the current sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
current→get_advertisedValue() Returns the current value of the current sensor (no more than 6 characters).
current→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number.
current→get_currentValue() Returns the current value of the current, in mA, as a floating point number.
current→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
current→get_errorMessage() Returns the error message of the latest error with the current sensor.
current→get_errorType()

Returns the numerical error code of the latest error with the current sensor.

current→get_friendlyName()

Returns a global identifier of the current sensor in the format `MODULE_NAME.FUNCTION_NAME`.

current→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

current→get_functionId()

Returns the hardware identifier of the current sensor, without reference to the module.

current→get_hardwareId()

Returns the unique hardware identifier of the current sensor in the form `SERIAL.FUNCTIONID`.

current→get_highestValue()

Returns the maximal value observed for the current since the device was started.

current→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

Returns the minimal value observed for the current since the device was started.

current→get_module()

Gets the `YModule` object for the device on which the function is located.

current→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

current→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

current→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

current→get_resolution()

Returns the resolution of the measured values.

current→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

current→get_unit()

Returns the measuring unit for the current.

current→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

current→isOnline()

Checks if the current sensor is currently reachable, without raising any error.

current→isOnline_async(**callback**, **context**)

Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

current→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

current→load(**msValidity**)

Preloads the current sensor cache with a specified validity duration.

current→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

current→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

current→load_async(msValidity, callback, context)

Preloads the current sensor cache with a specified validity duration (asynchronous version).

current→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

current→nextCurrent()

Continues the enumeration of current sensors started using `yFirstCurrent()`.

current→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

current→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

current→set_highestValue(newval)

Changes the recorded maximal value observed.

current→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

current→set_logicalName(newval)

Changes the logical name of the current sensor.

current→set_lowestValue(newval)

Changes the recorded minimal value observed.

current→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

current→set_resolution(newval)

Changes the resolution of the measured physical values.

current→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

current→startDataLogger()

Starts the data logger on the device.

current→stopDataLogger()

Stops the datalogger on the device.

current→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

current→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YCurrent.FindCurrent() YCurrent yFindCurrent()yFindCurrent()

Retrieves a current sensor for a given identifier.

```
function FindCurrent( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the current sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrent.isOnline()` to test if the current sensor is indeed online at a given time. In case of ambiguity when looking for a current sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the current sensor

Returns :

a `YCurrent` object allowing you to drive the current sensor.

YCurrent.FindCurrentInContext() YCurrent yFindCurrentInContext()

Retrieves a current sensor for a given identifier in a YAPI context.

function **FindCurrentInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the current sensor

Returns :

a `YCurrent` object allowing you to drive the current sensor.

YCurrent.FirstCurrent() YCurrent yFirstCurrent()yFirstCurrent()

Starts the enumeration of current sensors currently accessible.

```
function FirstCurrent( )
```

Use the method `YCurrent.nextCurrent()` to iterate on next current sensors.

Returns :

a pointer to a `YCurrent` object, corresponding to the first current sensor currently online, or a `null` pointer if there are none.

YCurrent.FirstCurrentInContext() YCurrent
yFirstCurrentInContext()

Starts the enumeration of current sensors currently accessible.

function **FirstCurrentInContext**(**yctx**)

Use the method `YCurrent.nextCurrent()` to iterate on next current sensors.

Parameters :

yctx a YAPI context.

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() YCurrent
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 linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→clearCache()current.clearCache() YCurrent

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the current sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

current→describe()current.describe() YCurrent

Returns a short text that describes unambiguously the instance of the current sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the current sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

current→get_advertisedValue() YCurrent
current→advertisedValue()
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() YCurrent
current→currentRawValue()
current.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

current→get_currentValue() YCurrent
current→currentValue()current.get_currentValue()

Returns the current value of the current, in mA, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the current, in mA, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

current→get_dataLogger() YCurrent
current→dataLogger()current.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

current→get_errorMessage() YCurrent
current→errorMessage()current.get_errorMessage()

Returns the error message of the latest error with the current sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the current sensor object

current→get_errorType() YCurrent
current→errorType()current.get_errorType()

Returns the numerical error code of the latest error with the current sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the current sensor object

current→get_friendlyName() YCurrent
current→friendlyName()current.get_friendlyName()

Returns a global identifier of the current sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the current sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the current sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the current sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

current→get_functionDescriptor() YCurrent
current→functionDescriptor()
current.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

current→get_functionId() YCurrent
current→functionId()current.get_functionId()

Returns the hardware identifier of the current sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the current sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

current→get_hardwareId() YCurrent
current→hardwareId()current.get_hardwareId()

Returns the unique hardware identifier of the current sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the current sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the current sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

current→get_highestValue() YCurrent
current→highestValue()current.get_highestValue()

Returns the maximal value observed for the current since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the current since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

current→get_logFrequency() YCurrent
current→logFrequency()current.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

current→get_logicalName() YCurrent
current→logicalName()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() YCurrent
current→lowestValue()current.get_lowestValue()

Returns the minimal value observed for the current since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the current since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

current→get_module() YCurrent
current→module()current.get_module()

Gets the YModule object for the device on which the function is located.

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_recordedData() YCurrent
current→recordedData()current.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

current→get_reportFrequency() YCurrent
current→reportFrequency()
current.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

current→get_resolution() YCurrent
current→resolution()current.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

current→get_sensorState() YCurrent
current→sensorState()current.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

current→get_unit() YCurrent
current→unit()current.get_unit()

Returns the measuring unit for the current.

```
function get_unit( )
```

Returns :
a string corresponding to the measuring unit for the current

On failure, throws an exception or returns `Y_UNIT_INVALID`.

current→get_userData() YCurrent
current→userData()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()current.isOnline() YCurrent

Checks if the current sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the current sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the current sensor.

Returns :

`true` if the current sensor can be reached, and `false` otherwise

current→load()current.load() YCurrent

Preloads the current sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()current.loadAttribute() YCurrent

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

current→loadCalibrationPoints() YCurrent
current.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→muteValueCallbacks() YCurrent
current.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

current→nextCurrent()current.nextCurrent() YCurrent

Continues the enumeration of current sensors started using `yFirstCurrent()`.

function **nextCurrent**( )

Returns :

a pointer to a `YCurrent` object, corresponding to a current sensor currently online, or a `null` pointer if there are no more current sensors to enumerate.

current→registerTimedReportCallback() YCurrent
current.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

current→registerValueCallback() YCurrent
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() YCurrent
current→setHighestValue()
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_logFrequency() YCurrent
current→setLogFrequency()
current.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_logicalName() YCurrent
current→setLogicalName()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() YCurrent
current→setLowestValue()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_reportFrequency() YCurrent
current→setReportFrequency()
current.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_resolution() YCurrent
current→setResolution()current.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_userData() YCurrent
current→setUserData()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

current→startDataLogger()current.startDataLogger() YCurrent

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

current→stopDataLogger()current.stopDataLogger() YCurrent

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

current→unmuteValueCallbacks() YCurrent
current.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

current→wait_async()current.wait_async() YCurrent

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.15. CurrentLoopOutput function interface

The Yoctopuce application programming interface allows you to change the value of the 4-20mA output as well as to know the current loop state.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_currentloopoutput.js'></script>
cpp	#include "yocto_currentloopoutput.h"
m	#import "yocto_currentloopoutput.h"
pas	uses yocto_currentloopoutput;
vb	yocto_currentloopoutput.vb
cs	yocto_currentloopoutput.cs
java	import com.yoctopuce.YoctoAPI.YCurrentLoopOutput;
uwp	import com.yoctopuce.YoctoAPI.YCurrentLoopOutput;
py	from yocto_currentloopoutput import *
php	require_once('yocto_currentloopoutput.php');
es	in HTML: <script src="../../lib/yocto_currentloopoutput.js"></script> in node.js: require('yoctolib-es2017/yocto_currentloopoutput.js');

Global functions

yFindCurrentLoopOutput(**func**)

Retrieves a 4-20mA output for a given identifier.

yFindCurrentLoopOutputInContext(**yctx**, **func**)

Retrieves a 4-20mA output for a given identifier in a YAPI context.

yFirstCurrentLoopOutput()

Starts the enumeration of 4-20mA outputs currently accessible.

yFirstCurrentLoopOutputInContext(**yctx**)

Starts the enumeration of 4-20mA outputs currently accessible.

`YCurrentLoopOutput` methods

currentloopoutput→clearCache()

Invalidates the cache.

currentloopoutput→currentMove(**mA_target**, **ms_duration**)

Performs a smooth transistion of current flowing in the loop.

currentloopoutput→describe()

Returns a short text that describes unambiguously the instance of the 4-20mA output in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

currentloopoutput→get_advertisedValue()

Returns the current value of the 4-20mA output (no more than 6 characters).

currentloopoutput→get_current()

Returns the loop current set point in mA.

currentloopoutput→get_currentAtStartUp()

Returns the current in the loop at device startup, in mA.

currentloopoutput→get_errorMessage()

Returns the error message of the latest error with the 4-20mA output.

currentloopoutput→get_errorType()

Returns the numerical error code of the latest error with the 4-20mA output.

currentloopoutput→get_friendlyName()

Returns a global identifier of the 4-20mA output in the format `MODULE_NAME.FUNCTION_NAME`.

currentloopoutput→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

currentloopoutput→get_functionId()

Returns the hardware identifier of the 4-20mA output, without reference to the module.

currentloopoutput→get_hardwareId()

Returns the unique hardware identifier of the 4-20mA output in the form SERIAL.FUNCTIONID.

currentloopoutput→get_logicalName()

Returns the logical name of the 4-20mA output.

currentloopoutput→get_loopPower()

Returns the loop powerstate.

currentloopoutput→get_module()

Gets the YModule object for the device on which the function is located.

currentloopoutput→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

currentloopoutput→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

currentloopoutput→isOnline()

Checks if the 4-20mA output is currently reachable, without raising any error.

currentloopoutput→isOnline_async(**callback**, **context**)

Checks if the 4-20mA output is currently reachable, without raising any error (asynchronous version).

currentloopoutput→load(**msValidity**)

Preloads the 4-20mA output cache with a specified validity duration.

currentloopoutput→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

currentloopoutput→load_async(**msValidity**, **callback**, **context**)

Preloads the 4-20mA output cache with a specified validity duration (asynchronous version).

currentloopoutput→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

currentloopoutput→nextCurrentLoopOutput()

Continues the enumeration of 4-20mA outputs started using yFirstCurrentLoopOutput().

currentloopoutput→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

currentloopoutput→set_current(**newval**)

Changes the current loop, the valid range is from 3 to 21mA.

currentloopoutput→set_currentAtStartUp(**newval**)

Changes the loop current at device start up.

currentloopoutput→set_logicalName(**newval**)

Changes the logical name of the 4-20mA output.

currentloopoutput→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

currentloopoutput→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

currentloopoutput→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YCurrentLoopOutput.FindCurrentLoopOutput() YCurrentLoopOutput yFindCurrentLoopOutput()yFindCurrentLoopOutput()

Retrieves a 4-20mA output for a given identifier.

```
function FindCurrentLoopOutput( 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 4-20mA output is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrentLoopOutput.isOnline()` to test if the 4-20mA output is indeed online at a given time. In case of ambiguity when looking for a 4-20mA output 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the 4-20mA output

Returns :

a `YCurrentLoopOutput` object allowing you to drive the 4-20mA output.

YCurrentLoopOutput.FindCurrentLoopOutputInContext() yFindCurrentLoopOutputInContext()

YCurrentLoopOutput

Retrieves a 4-20mA output for a given identifier in a YAPI context.

function **FindCurrentLoopOutputInContext**(**yctx**, **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 4-20mA output is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrentLoopOutput.isOnline()` to test if the 4-20mA output is indeed online at a given time. In case of ambiguity when looking for a 4-20mA output 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 :

yctx a YAPI context

func a string that uniquely characterizes the 4-20mA output

Returns :

a `YCurrentLoopOutput` object allowing you to drive the 4-20mA output.

YCurrentLoopOutput.FirstCurrentLoopOutput() YCurrentLoopOutput yFirstCurrentLoopOutput() yFirstCurrentLoopOutput()

Starts the enumeration of 4-20mA outputs currently accessible.

function **FirstCurrentLoopOutput**( )

Use the method `YCurrentLoopOutput.nextCurrentLoopOutput()` to iterate on next 4-20mA outputs.

Returns :

a pointer to a `YCurrentLoopOutput` object, corresponding to the first 4-20mA output currently online, or a `null` pointer if there are none.

YCurrentLoopOutput.FirstCurrentLoopOutputInContext() yFirstCurrentLoopOutputInContext()

YCurrentLoopOutput

Starts the enumeration of 4-20mA outputs currently accessible.

function **FirstCurrentLoopOutputInContext**(**yctx**)

Use the method `YCurrentLoopOutput.nextCurrentLoopOutput()` to iterate on next 4-20mA outputs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YCurrentLoopOutput` object, corresponding to the first 4-20mA output currently online, or a `null` pointer if there are none.

currentloopoutput→clearCache() YCurrentLoopOutput
currentloopoutput.clearCache()

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the 4-20mA output attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

currentloopoutput→currentMove() YCurrentLoopOutput
currentloopoutput.currentMove()

Performs a smooth transistion of current flowing in the loop.

function **currentMove**(**mA_target**, **ms_duration**)

Any current explicit change cancels any ongoing transition process.

Parameters :

mA_target	new current value at the end of the transition (floating-point number, representing the end current in mA)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

currentloopoutput→describe() YCurrentLoopOutput
currentloopoutput.describe()

Returns a short text that describes unambiguously the instance of the 4-20mA output in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the 4-20mA output (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

currentloopoutput→get_advertisedValue() YCurrentLoopOutput
currentloopoutput→advertisedValue()
currentloopoutput.get_advertisedValue()

Returns the current value of the 4-20mA output (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the 4-20mA output (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

currentloopoutput→get_current() YCurrentLoopOutput
currentloopoutput→current()
currentloopoutput.get_current()

Returns the loop current set point in mA.

function **get_current**( )

Returns :
a floating point number corresponding to the loop current set point in mA

On failure, throws an exception or returns `Y_CURRENT_INVALID`.

currentloopoutput→get_currentAtStartUp() YCurrentLoopOutput
currentloopoutput→currentAtStartUp()
currentloopoutput.get_currentAtStartUp()

Returns the current in the loop at device startup, in mA.

function **get_currentAtStartUp**( )

Returns :
a floating point number corresponding to the current in the loop at device startup, in mA

On failure, throws an exception or returns `Y_CURRENTATSTARTUP_INVALID`.

currentloopoutput→get_errorMessage() YCurrentLoopOutput
currentloopoutput→errorMessage()
currentloopoutput.get_errorMessage()

Returns the error message of the latest error with the 4-20mA output.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the 4-20mA output object

currentloopoutput→get_errorType() YCurrentLoopOutput
currentloopoutput→errorType()
currentloopoutput.get_errorType()

Returns the numerical error code of the latest error with the 4-20mA output.

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 the 4-20mA output object

currentloopoutput→get_friendlyName() YCurrentLoopOutput
currentloopoutput→friendlyName()
currentloopoutput.get_friendlyName()

Returns a global identifier of the 4-20mA output in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the 4-20mA output if they are defined, otherwise the serial number of the module and the hardware identifier of the 4-20mA output (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the 4-20mA output using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

currentloopoutput→get_functionDescriptor() YCurrentLoopOutput
currentloopoutput→functionDescriptor()
currentloopoutput.get_functionDescriptor()

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.

currentloopoutput→get_functionId() YCurrentLoopOutput
currentloopoutput→functionId()
currentloopoutput.get_functionId()

Returns the hardware identifier of the 4-20mA output, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the 4-20mA output (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

currentloopoutput→get_hardwareId() YCurrentLoopOutput
currentloopoutput→hardwareId()
currentloopoutput.get_hardwareId()

Returns the unique hardware identifier of the 4-20mA output in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the 4-20mA output (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the 4-20mA output (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

currentloopoutput→get_logicalName() YCurrentLoopOutput
currentloopoutput→logicalName()
currentloopoutput.get_logicalName()

Returns the logical name of the 4-20mA output.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the 4-20mA output.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

currentloopoutput→get_loopPower() YCurrentLoopOutput
currentloopoutput→loopPower()
currentloopoutput.get_loopPower()

Returns the loop powerstate.

function **get_loopPower**( )

POWEROK: the loop is powered. NOPWR: the loop in not powered. LOWPWR: the loop is not powered enough to maintain the current required (insufficient voltage).

Returns :
a value among Y_LOOPPOWER_NOPWR, Y_LOOPPOWER_LOWPWR and Y_LOOPPOWER_POWEROK corresponding to the loop powerstate

On failure, throws an exception or returns Y_LOOPPOWER_INVALID.

currentloopoutput→get_module() YCurrentLoopOutput
currentloopoutput→module()
currentloopoutput.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

currentloopoutput→get_userData() YCurrentLoopOutput
currentloopoutput→userData()
currentloopoutput.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.

currentloopoutput→isOnline() YCurrentLoopOutput
currentloopoutput.isOnline()

Checks if the 4-20mA output is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the 4-20mA output 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 4-20mA output.

Returns :

`true` if the 4-20mA output can be reached, and `false` otherwise

currentloopoutput→load()currentloopoutput.load() YCurrentLoopOutput

Preloads the 4-20mA output 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.

currentloopoutput→loadAttribute() YCurrentLoopOutput
currentloopoutput.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

currentloopoutput→muteValueCallbacks() YCurrentLoopOutput
currentloopoutput.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

currentloopoutput→nextCurrentLoopOutput() YCurrentLoopOutput
currentloopoutput.nextCurrentLoopOutput()

Continues the enumeration of 4-20mA outputs started using `yFirstCurrentLoopOutput()`.

function **nextCurrentLoopOutput**( )

Returns :

a pointer to a `YCurrentLoopOutput` object, corresponding to a 4-20mA output currently online, or a `null` pointer if there are no more 4-20mA outputs to enumerate.

currentloopoutput→registerValueCallback() YCurrentLoopOutput
currentloopoutput.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.

currentloopoutput→set_current() YCurrentLoopOutput
currentloopoutput→setCurrent()
currentloopoutput.set_current()

Changes the current loop, the valid range is from 3 to 21mA.

function **set_current**(**newval**)

If the loop is not propely powered, the target current is not reached and loopPower is set to LOWPWR.

Parameters :

newval a floating point number corresponding to the current loop, the valid range is from 3 to 21mA

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

currentloopoutput→set_currentAtStartUp() YCurrentLoopOutput
currentloopoutput→setCurrentAtStartUp()
currentloopoutput.set_currentAtStartUp()

Changes the loop current at device start up.

function **set_currentAtStartUp**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call has no effect.

Parameters :

newval a floating point number corresponding to the loop current at device start up

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

currentloopoutput→set_logicalName() YCurrentLoopOutput
currentloopoutput→setLogicalName()
currentloopoutput.set_logicalName()

Changes the logical name of the 4-20mA output.

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 4-20mA output.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

currentloopoutput→set_userData() YCurrentLoopOutput
currentloopoutput→setUserData()
currentloopoutput.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

currentloopoutput→unmuteValueCallbacks() YCurrentLoopOutput
currentloopoutput.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

currentloopoutput→wait_async() YCurrentLoopOutput
currentloopoutput.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.16. DaisyChain function interface

The YDaisyChain interface can be used to verify that devices that are daisy-chained directly from device to device, without a hub, are detected properly.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_daisychain.js'></script>
cpp	#include "yocto_daisychain.h"
m	#import "yocto_daisychain.h"
pas	uses yocto_daisychain;
vb	yocto_daisychain.vb
cs	yocto_daisychain.cs
java	import com.yoctopuce.YoctoAPI.YDaisyChain;
uwp	import com.yoctopuce.YoctoAPI.YDaisyChain;
py	from yocto_daisychain import *
php	require_once('yocto_daisychain.php');
es	in HTML: <script src="../../lib/yocto_daisychain.js"></script> in node.js: require('yoctolib-es2017/yocto_daisychain.js');

Global functions

yFindDaisyChain(**func**)

Retrieves a module chain for a given identifier.

yFindDaisyChainInContext(**yctx**, **func**)

Retrieves a module chain for a given identifier in a YAPI context.

yFirstDaisyChain()

Starts the enumeration of module chains currently accessible.

yFirstDaisyChainInContext(**yctx**)

Starts the enumeration of module chains currently accessible.

YDaisyChain methods

daisychain→clearCache()

Invalidates the cache.

daisychain→describe()

Returns a short text that describes unambiguously the instance of the module chain in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

daisychain→get_advertisedValue()

Returns the current value of the module chain (no more than 6 characters).

daisychain→get_childCount()

Returns the number of child nodes currently detected.

daisychain→get_daisyState()

Returns the state of the daisy-link between modules.

daisychain→get_errorMessage()

Returns the error message of the latest error with the module chain.

daisychain→get_errorType()

Returns the numerical error code of the latest error with the module chain.

daisychain→get_friendlyName()

Returns a global identifier of the module chain in the format `MODULE_NAME.FUNCTION_NAME`.

daisychain→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

daisychain→get_functionId()

Returns the hardware identifier of the module chain, without reference to the module.

daisychain→get_hardwareId()

Returns the unique hardware identifier of the module chain in the form `SERIAL.FUNCTIONID`.

daisychain→get_logicalName()

Returns the logical name of the module chain.

daisychain→get_module()

Gets the `YModule` object for the device on which the function is located.

daisychain→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

daisychain→get_requiredChildCount()

Returns the number of child nodes expected in normal conditions.

daisychain→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

daisychain→isOnline()

Checks if the module chain is currently reachable, without raising any error.

daisychain→isOnline_async(callback, context)

Checks if the module chain is currently reachable, without raising any error (asynchronous version).

daisychain→load(msValidity)

Preloads the module chain cache with a specified validity duration.

daisychain→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

daisychain→load_async(msValidity, callback, context)

Preloads the module chain cache with a specified validity duration (asynchronous version).

daisychain→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

daisychain→nextDaisyChain()

Continues the enumeration of module chains started using `yFirstDaisyChain()`.

daisychain→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

daisychain→set_logicalName(newval)

Changes the logical name of the module chain.

daisychain→set_requiredChildCount(newval)

Changes the number of child nodes expected in normal conditions.

daisychain→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

daisychain→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

daisychain→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDaisyChain.FindDaisyChain() YDaisyChain yFindDaisyChain()yFindDaisyChain()

Retrieves a module chain for a given identifier.

```
function FindDaisyChain( 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 module chain is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDaisyChain.isOnline()` to test if the module chain is indeed online at a given time. In case of ambiguity when looking for a module chain 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the module chain

Returns :

a `YDaisyChain` object allowing you to drive the module chain.

YDaisyChain.FindDaisyChainInContext() yFindDaisyChainInContext()

YDaisyChain

Retrieves a module chain for a given identifier in a YAPI context.

function **FindDaisyChainInContext**(**yctx**, **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 module chain is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDaisyChain.isOnline()` to test if the module chain is indeed online at a given time. In case of ambiguity when looking for a module chain 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 :

yctx a YAPI context

func a string that uniquely characterizes the module chain

Returns :

a `YDaisyChain` object allowing you to drive the module chain.

YDaisyChain.FirstDaisyChain() YDaisyChain yFirstDaisyChain()yFirstDaisyChain()

Starts the enumeration of module chains currently accessible.

function **FirstDaisyChain**( )

Use the method `YDaisyChain.nextDaisyChain()` to iterate on next module chains.

Returns :

a pointer to a `YDaisyChain` object, corresponding to the first module chain currently online, or a `null` pointer if there are none.

YDaisyChain.FirstDaisyChainInContext() YDaisyChain yFirstDaisyChainInContext()

Starts the enumeration of module chains currently accessible.

function **FirstDaisyChainInContext**(**yctx**)

Use the method `YDaisyChain.nextDaisyChain()` to iterate on next module chains.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDaisyChain` object, corresponding to the first module chain currently online, or a `null` pointer if there are none.

daisychain→clearCache()daisychain.clearCache() YDaisyChain

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the module chain attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

daisychain→describe()daisychain.describe() YDaisyChain

Returns a short text that describes unambiguously the instance of the module chain in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the module chain (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

daisychain→get_advertisedValue() YDaisyChain
daisychain→advertisedValue()
daisychain.get_advertisedValue()

Returns the current value of the module chain (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the module chain (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

daisychain→get_childCount() YDaisyChain
daisychain→childCount()
daisychain.get_childCount()

Returns the number of child nodes currently detected.

function **get_childCount**( )

Returns :
an integer corresponding to the number of child nodes currently detected

On failure, throws an exception or returns `Y_CHILDCOUNT_INVALID`.

daisychain→get_daisyState() YDaisyChain
daisychain→daisyState()daisychain.get_daisyState()

Returns the state of the daisy-link between modules.

function **get_daisyState**( )

Returns :

a value among Y_DAISYSTATE_READY, Y_DAISYSTATE_IS_CHILD, Y_DAISYSTATE_FIRMWARE_MISMATCH, Y_DAISYSTATE_CHILD_MISSING and Y_DAISYSTATE_CHILD_LOST corresponding to the state of the daisy-link between modules

On failure, throws an exception or returns Y_DAISYSTATE_INVALID.

daisychain→get_errorMessage() YDaisyChain
daisychain→errorMessage()
daisychain.get_errorMessage()

Returns the error message of the latest error with the module chain.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the module chain object

daisychain→get_errorType() YDaisyChain
daisychain→errorType()daisychain.get_errorType()

Returns the numerical error code of the latest error with the module chain.

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 the module chain object

daisychain→get_friendlyName() YDaisyChain
daisychain→friendlyName()
daisychain.get_friendlyName()

Returns a global identifier of the module chain in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the module chain if they are defined, otherwise the serial number of the module and the hardware identifier of the module chain (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the module chain using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

daisychain→get_functionDescriptor() YDaisyChain
daisychain→functionDescriptor()
daisychain.get_functionDescriptor()

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.

daisychain→get_functionId() YDaisyChain
daisychain→functionId()daisychain.get_functionId()

Returns the hardware identifier of the module chain, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the module chain (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

daisychain→get_hardwareId() YDaisyChain
daisychain→hardwareId()
daisychain.get_hardwareId()

Returns the unique hardware identifier of the module chain in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the module chain (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the module chain (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

daisychain→get_logicalName() YDaisyChain
daisychain→logicalName()
daisychain.get_logicalName()

Returns the logical name of the module chain.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the module chain.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

daisychain→get_module() YDaisyChain
daisychain→module()daisychain.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

daisychain→get_requiredChildCount() YDaisyChain
daisychain→requiredChildCount()
daisychain.get_requiredChildCount()

Returns the number of child nodes expected in normal conditions.

function **get_requiredChildCount**( )

Returns :
an integer corresponding to the number of child nodes expected in normal conditions

On failure, throws an exception or returns `Y_REQUIREDCHILDCOUNT_INVALID`.

daisychain→get_userData() YDaisyChain
daisychain→userData()daisychain.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.

daisychain→isOnline()daisychain.isOnline() YDaisyChain

Checks if the module chain is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the module chain 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 module chain.

Returns :

`true` if the module chain can be reached, and `false` otherwise

daisychain→load()daisychain.load() YDaisyChain

Preloads the module chain 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.

daisychain→loadAttribute()daisychain.loadAttribute() YDaisyChain

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

daisychain→muteValueCallbacks() YDaisyChain
daisychain.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

daisychain→nextDaisyChain() YDaisyChain
daisychain.nextDaisyChain()

Continues the enumeration of module chains started using `yFirstDaisyChain()`.

function **nextDaisyChain**( )

Returns :

a pointer to a `YDaisyChain` object, corresponding to a module chain currently online, or a `null` pointer if there are no more module chains to enumerate.

daisychain→registerValueCallback() YDaisyChain
daisychain.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.

daisychain→set_logicalName() YDaisyChain
daisychain→setLogicalName()
daisychain.set_logicalName()

Changes the logical name of the module chain.

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 chain.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

daisychain→set_requiredChildCount() YDaisyChain
daisychain→setRequiredChildCount()
daisychain.set_requiredChildCount()

Changes the number of child nodes expected in normal conditions.

function **set_requiredChildCount**(**newval**)

If the value is zero, no check is performed. If it is non-zero, the number child nodes is checked on startup and the status will change to error if the count does not match.

Parameters :

newval an integer corresponding to the number of child nodes expected in normal conditions

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

daisychain→set_userData() YDaisyChain
daisychain→setUserData()daisychain.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

daisychain→unmuteValueCallbacks() YDaisyChain
daisychain.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

daisychain→wait_async()daisychain.wait_async() YDaisyChain

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.17. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
uwp     import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
php     require_once('yocto_api.php');
es      in HTML: <script src="../../lib/yocto_api.js"></script>
        in node.js: require('yoctolib-es2017/yocto_api.js');
```

Global functions
yFindDataLogger(**func**) Retrieves a data logger for a given identifier.
yFindDataLoggerInContext(**yctx**, **func**) Retrieves a data logger for a given identifier in a YAPI context.
yFirstDataLogger() Starts the enumeration of data loggers currently accessible.
yFirstDataLoggerInContext(**yctx**) Starts the enumeration of data loggers currently accessible.
`YDataLogger` methods
datalogger→clearCache() Invalidates the cache.
datalogger→describe() Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
datalogger→forgetAllDataStreams() Clears the data logger memory and discards all recorded data streams.
datalogger→get_advertisedValue() Returns the current value of the data logger (no more than 6 characters).
datalogger→get_autoStart() Returns the default activation state of the data logger on power up.
datalogger→get_beaconDriven() Returns true if the data logger is synchronised with the localization beacon.
datalogger→get_currentRunIndex() Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.
datalogger→get_dataSets() Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.
datalogger→get_dataStreams(**v**)

Builds a list of all data streams hold by the data logger (legacy method).

datalogger→get_errorMessage()
Returns the error message of the latest error with the data logger.

datalogger→get_errorType()
Returns the numerical error code of the latest error with the data logger.

datalogger→get_friendlyName()
Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

datalogger→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

datalogger→get_functionId()
Returns the hardware identifier of the data logger, without reference to the module.

datalogger→get_hardwareId()
Returns the unique hardware identifier of the data logger in the form `SERIAL.FUNCTIONID`.

datalogger→get_logicalName()
Returns the logical name of the data logger.

datalogger→get_module()
Gets the `YModule` object for the device on which the function is located.

datalogger→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

datalogger→get_recording()
Returns the current activation state of the data logger.

datalogger→get_timeUTC()
Returns the Unix timestamp for current UTC time, if known.

datalogger→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

datalogger→isOnline()
Checks if the data logger is currently reachable, without raising any error.

datalogger→isOnline_async(**callback**, **context**)
Checks if the data logger is currently reachable, without raising any error (asynchronous version).

datalogger→load(**msValidity**)
Preloads the data logger cache with a specified validity duration.

datalogger→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

datalogger→load_async(**msValidity**, **callback**, **context**)
Preloads the data logger cache with a specified validity duration (asynchronous version).

datalogger→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

datalogger→nextDataLogger()
Continues the enumeration of data loggers started using `yFirstDataLogger()`.

datalogger→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

datalogger→set_autoStart(**newval**)
Changes the default activation state of the data logger on power up.

datalogger→set_beaconDriven(**newval**)

Changes the type of synchronisation of the data logger.

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

datalogger→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

datalogger→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDataLogger.FindDataLogger() YDataLogger yFindDataLogger()yFindDataLogger()

Retrieves a data logger for a given identifier.

function **FindDataLogger**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FindDataLoggerInContext() YDataLogger yFindDataLoggerInContext()

Retrieves a data logger for a given identifier in a YAPI context.

function **FindDataLoggerInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger() YDataLogger yFirstDataLogger()yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

function **FirstDataLogger**( )

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

YDataLogger.FirstDataLoggerInContext() YDataLogger yFirstDataLoggerInContext()

Starts the enumeration of data loggers currently accessible.

function **FirstDataLoggerInContext**(**yctx**)

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

datalogger→clearCache()datalogger.clearCache() YDataLogger

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the data logger attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

datalogger→describe()datalogger.describe() YDataLogger

Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the data logger (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

datalogger→forgetAllDataStreams() YDataLogger
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() YDataLogger
datalogger→advertisedValue()
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() YDataLogger
datalogger→autoStart()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_beaconDriven() YDataLogger
datalogger→beaconDriven()
datalogger.get_beaconDriven()

Returns true if the data logger is synchronised with the localization beacon.

function **get_beaconDriven**( )

Returns :

either Y_BEACONDRIVEN_OFF or Y_BEACONDRIVEN_ON, according to true if the data logger is synchronised with the localization beacon

On failure, throws an exception or returns Y_BEACONDRIVEN_INVALID.

datalogger→get_currentRunIndex() YDataLogger
datalogger→currentRunIndex()
datalogger.get_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

function **get_currentRunIndex**( )

Returns :

an integer corresponding to the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point

On failure, throws an exception or returns `Y_CURRENTRUNINDEX_INVALID`.

datalogger→get_dataSets() YDataLogger
datalogger→dataSets()datalogger.get_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

function **get_dataSets**( )

This function only works if the device uses a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

Returns :
a list of YDataSet object.

On failure, throws an exception or returns an empty list.

datalogger→get_dataStreams() YDataLogger
datalogger→dataStreams()
datalogger.get_dataStreams()

Builds a list of all data streams hold by the data logger (legacy method).

function **get_dataStreams**(**v**)

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

v an array of YDataStream objects to be filled in

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→get_errorMessage() YDataLogger
datalogger→errorMessage()
datalogger.get_errorMessage()

Returns the error message of the latest error with the data logger.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the data logger object

datalogger→get_errorType() YDataLogger
datalogger→errorType()datalogger.get_errorType()

Returns the numerical error code of the latest error with the data logger.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the data logger object

datalogger→get_friendlyName() YDataLogger
datalogger→friendlyName()
datalogger.get_friendlyName()

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the data logger using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

datalogger→get_functionDescriptor() YDataLogger
datalogger→functionDescriptor()
datalogger.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

datalogger→get_functionId() YDataLogger
datalogger→functionId()datalogger.get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the data logger (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

datalogger→get_hardwareId() YDataLogger
datalogger→hardwareId()
datalogger.get_hardwareId()

Returns the unique hardware identifier of the data logger in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the data logger (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

datalogger→get_logicalName() YDataLogger
datalogger→logicalName()
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`.

datalogger→get_module() YDataLogger
datalogger→module()datalogger.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

datalogger→get_recording() YDataLogger
datalogger→recording()datalogger.get_recording()

Returns the current activation state of the data logger.

function **get_recording**( )

Returns :

a value among `Y_RECORDING_OFF`, `Y_RECORDING_ON` and `Y_RECORDING_PENDING` corresponding to the current activation state of the data logger

On failure, throws an exception or returns `Y_RECORDING_INVALID`.

datalogger→get_timeUTC() YDataLogger
datalogger→timeUTC()datalogger.get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

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() YDataLogger
datalogger→userData()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()datalogger.isOnline() YDataLogger

Checks if the data logger is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

Returns :
`true` if the data logger can be reached, and `false` otherwise

datalogger→load()datalogger.load() YDataLogger

Preloads the data logger cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()datalogger.loadAttribute() YDataLogger

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

datalogger→muteValueCallbacks() YDataLogger
datalogger.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→nextDataLogger() YDataLogger
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() YDataLogger
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() YDataLogger
datalogger→setAutoStart()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_beaconDriven() YDataLogger
datalogger→setBeaconDriven()
datalogger.set_beaconDriven()

Changes the type of synchronisation of the data logger.

function **set_beaconDriven**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to the type of synchronisation of the data logger

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_logicalName() YDataLogger
datalogger→setLogicalName()
datalogger.set_logicalName()

Changes the logical name of the data logger.

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() YDataLogger
datalogger→setRecording()
datalogger.set_recording()

Changes the activation state of the data logger to start/stop recording data.

function **set_recording**(**newval**)

Parameters :

newval a value among `Y_RECORDING_OFF`, `Y_RECORDING_ON` and `Y_RECORDING_PENDING` corresponding to the activation state of the data logger to start/stop recording data

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_timeUTC() YDataLogger
datalogger→setTimeUTC()datalogger.set_timeUTC()

Changes the current UTC time reference used for recorded data.

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() YDataLogger
datalogger→setUserData()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

datalogger→unmuteValueCallbacks() YDataLogger
datalogger.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→wait_async()datalogger.wait_async() YDataLogger

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.18. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_datalogger.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDataLogger = yoctolib.YDataLogger;
php     require_once('yocto_datalogger.php');
cpp     #include "yocto_datalogger.h"
m       #import "yocto_datalogger.h"
pas     uses yocto_datalogger;
vb      yocto_datalogger.vb
cs      yocto_datalogger.cs
java    import com.yoctopuce.YoctoAPI.YDataLogger;
py      from yocto_datalogger import *
```

YDataRun methods
datarun→get_averageValue(measureName, pos) Returns the average value of the measure observed at the specified time period.
datarun→get_duration() Returns the duration (in seconds) of the data run.
datarun→get_maxValue(measureName, pos) Returns the maximal value of the measure observed at the specified time period.
datarun→get_measureNames() Returns the names of the measures recorded by the data logger.
datarun→get_minValue(measureName, pos) Returns the minimal value of the measure observed at the specified time period.
datarun→get_startTimeUTC() Returns the start time of the data run, relative to the Jan 1, 1970.
datarun→get_valueCount() Returns the number of values accessible in this run, given the selected data samples interval.
datarun→get_valueInterval() Returns the number of seconds covered by each value in this run.
datarun→set_valueInterval(valueInterval) Changes the number of seconds covered by each value in this run.

datarun→get_startTimeUTC() YDataRun
datarun→startTimeUTC()

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).

3.19. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instantiated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');

YDataSet methods
dataset→get_endTimeUTC() Returns the end time of the dataset, relative to the Jan 1, 1970.
dataset→get_functionId() Returns the hardware identifier of the function that performed the measure, without reference to the module.
dataset→get_hardwareId() Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.
dataset→get_measures() Returns all measured values currently available for this DataSet, as a list of YMeasure objects.
dataset→get_measuresAt(**measure**) Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.
dataset→get_preview() Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.
dataset→get_progress() Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.
dataset→get_startTimeUTC() Returns the start time of the dataset, relative to the Jan 1, 1970.
dataset→get_summary()

Returns an YMeasure object which summarizes the whole DataSet.

dataset→get_unit()

Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(**callback**, **context**)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→get_endTimeUTC() YDataSet
dataset→endTimeUTC()dataset.get_endTimeUTC()

Returns the end time of the dataset, relative to the Jan 1, 1970.

function **get_endTimeUTC**( )

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

dataset→get_functionId() YDataSet
dataset→functionId()dataset.get_functionId()

Returns the hardware identifier of the function that performed the measure, without reference to the module.

function **get_functionId**( )

For example `temperature1`.

Returns :

a string that identifies the function (ex: `temperature1`)

dataset→get_hardwareId() YDataSet
dataset→hardwareId()dataset.get_hardwareId()

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example THRMCPL1-123456.temperature1)

Returns :

a string that uniquely identifies the function (ex: THRMCPL1-123456.temperature1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

dataset→get_measures() YDataSet
dataset→measures()dataset.get_measures()

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

function **get_measures**( )

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call loadMore() several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :
a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→get_measuresAt() YDataSet
dataset→measuresAt()dataset.get_measuresAt()

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

function **get_measuresAt**(**measure**)

The result is provided as a list of YMeasure objects.

Parameters :

measure condensed measure from the list previously returned by `get_preview()`.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_preview() YDataSet
dataset→preview()dataset.get_preview()

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

function **get_preview**( )

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :
a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_progress() YDataSet
dataset→progress()dataset.get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

```
function get_progress( )
```

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :
an integer in the range 0 to 100 (percentage of completion).

dataset→get_startTimeUTC() YDataSet
dataset→startTimeUTC()dataset.get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

function **get_startTimeUTC**( )

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

dataset→get_summary() YDataSet
dataset→summary()dataset.get_summary()

Returns an YMeasure object which summarizes the whole DataSet.

function **get_summary**( )

In includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

Returns :
an YMeasure object

dataset→get_unit() YDataSet
dataset→unit()dataset.get_unit()

Returns the measuring unit for the measured value.

function **get_unit**( )

Returns :
a string that represents a physical unit.

On failure, throws an exception or returns `Y_UNIT_INVALID`.

dataset→loadMore()dataset.loadMore() YDataSet

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

function **loadMore**( )

Returns :
an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

3.20. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');

YDataStream methods
datastream→get_averageValue() Returns the average of all measures observed within this stream.
datastream→get_columnCount() Returns the number of data columns present in this stream.
datastream→get_columnNames() Returns the title (or meaning) of each data column present in this stream.
datastream→get_data(**row**, **col**) Returns a single measure from the data stream, specified by its row and column index.
datastream→get_dataRows() Returns the whole data set contained in the stream, as a bidimensional table of numbers.
datastream→get_dataSamplesIntervalMs() Returns the number of milliseconds between two consecutive rows of this data stream.
datastream→get_duration() Returns the approximate duration of this stream, in seconds.
datastream→get_maxValue() Returns the largest measure observed within this stream.
datastream→get_minValue() Returns the smallest measure observed within this stream.
datastream→get_rowCount() Returns the number of data rows present in this stream.
datastream→get_runIndex() Returns the run index of the data stream.
datastream→get_startTime() Returns the relative start time of the data stream, measured in seconds.
datastream→get_startTimeUTC()

Returns the start time of the data stream, relative to the Jan 1, 1970.

datastream→get_averageValue() YDataStream
datastream→averageValue()
datastream.get_averageValue()

Returns the average of all measures observed within this stream.

function **get_averageValue**( )

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_columnCount() YDataStream
datastream→columnCount()
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()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

datastream→get_columnNames() YDataStream
datastream→columnNames()
datastream.get_columnNames()

Returns the title (or meaning) of each data column present in this stream.

function **get_columnNames**( )

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes _min, _avg and _max respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
a list containing as many strings as there are columns in the data stream.

On failure, throws an exception or returns an empty array.

datastream→get_data() YDataStream
datastream→data()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() YDataStream
datastream→dataRows()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_dataSamplesIntervalMs() YDataStream
datastream→dataSamplesIntervalMs()
datastream.get_dataSamplesIntervalMs()

Returns the number of milliseconds between two consecutive rows of this data stream.

function **get_dataSamplesIntervalMs**( )

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

Returns :
an unsigned number corresponding to a number of milliseconds.

datastream→get_duration() YDataStream
datastream→duration()datastream.get_duration()

Returns the approximate duration of this stream, in seconds.

function **get_duration**( )

Returns :
the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

datastream→get_maxValue() YDataStream
datastream→maxValue()datastream.get_maxValue()

Returns the largest measure observed within this stream.

function **get_maxValue**( )

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :
a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_minValue() YDataStream
datastream→minValue()datastream.get_minValue()

Returns the smallest measure observed within this stream.

function **get_minValue**( )

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the smallest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_rowCount() YDataStream
datastream→rowCount()datastream.get_rowCount()

Returns the number of data rows present in this stream.

function **get_rowCount**( )

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

datastream→get_runIndex() YDataStream
datastream→runIndex()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.

Returns :
an unsigned number corresponding to the run index.

datastream→get_startTime() YDataStream
datastream→startTime()datastream.get_startTime()

Returns the relative start time of the data stream, measured in seconds.

function **get_startTime**( )

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `get_startTimeUTC()`.

Returns :

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

datastream→get_startTimeUTC() YDataStream
datastream→startTimeUTC()
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.

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).

3.21. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_digitalio.js'></script>
cpp     #include "yocto_digitalio.h"
m       #import "yocto_digitalio.h"
pas     uses yocto_digitalio;
vb      yocto_digitalio.vb
cs      yocto_digitalio.cs
java    import com.yoctopuce.YoctoAPI.YDigitalIO;
uwp     import com.yoctopuce.YoctoAPI.YDigitalIO;
py      from yocto_digitalio import *
php     require_once('yocto_digitalio.php');
es      in HTML: <script src="../../lib/yocto_digitalio.js"></script>
        in node.js: require('yoctolib-es2017/yocto_digitalio.js');
```

Global functions
yFindDigitalIO(**func**) Retrieves a digital IO port for a given identifier.
yFindDigitalIOInContext(**yctx**, **func**) Retrieves a digital IO port for a given identifier in a YAPI context.
yFirstDigitalIO() Starts the enumeration of digital IO ports currently accessible.
yFirstDigitalIOInContext(**yctx**) Starts the enumeration of digital IO ports currently accessible.
`YDigitalIO` methods
digitalio→clearCache() Invalidates the cache.
digitalio→delayedPulse(**bitno**, **ms_delay**, **ms_duration**) Schedules a pulse on a single bit for a specified duration.
digitalio→describe() Returns a short text that describes unambiguously the instance of the digital IO port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
digitalio→get_advertisedValue() Returns the current value of the digital IO port (no more than 6 characters).
digitalio→get_bitDirection(**bitno**) Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).
digitalio→get_bitOpenDrain(**bitno**) Returns the type of electrical interface of a single bit from the I/O port.
digitalio→get_bitPolarity(**bitno**) Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).
digitalio→get_bitState(**bitno**) Returns the state of a single bit of the I/O port.

digitalio→get_errorMessage()
Returns the error message of the latest error with the digital IO port.

digitalio→get_errorType()
Returns the numerical error code of the latest error with the digital IO port.

digitalio→get_friendlyName()
Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

digitalio→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

digitalio→get_functionId()
Returns the hardware identifier of the digital IO port, without reference to the module.

digitalio→get_hardwareId()
Returns the unique hardware identifier of the digital IO port in the form `SERIAL.FUNCTIONID`.

digitalio→get_logicalName()
Returns the logical name of the digital IO port.

digitalio→get_module()
Gets the `YModule` object for the device on which the function is located.

digitalio→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

digitalio→get_outputVoltage()
Returns the voltage source used to drive output bits.

digitalio→get_portDiags()
Returns the port state diagnostics (Yocto-IO and Yocto-MaxiIO-V2 only).

digitalio→get_portDirection()
Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→get_portOpenDrain()
Returns the electrical interface for each bit of the port.

digitalio→get_portPolarity()
Returns the polarity of all the bits of the port.

digitalio→get_portSize()
Returns the number of bits implemented in the I/O port.

digitalio→get_portState()
Returns the digital IO port state: bit 0 represents input 0, and so on.

digitalio→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

digitalio→isOnline()
Checks if the digital IO port is currently reachable, without raising any error.

digitalio→isOnline_async(**callback**, **context**)
Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

digitalio→load(**msValidity**)
Preloads the digital IO port cache with a specified validity duration.

digitalio→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

digitalio→load_async(**msValidity**, **callback**, **context**)
Preloads the digital IO port cache with a specified validity duration (asynchronous version).

digitalio→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

digitalio→nextDigitalIO()

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

digitalio→pulse(**bitno**, **ms_duration**)

Triggers a pulse on a single bit for a specified duration.

digitalio→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

digitalio→set_bitDirection(**bitno**, **bitdirection**)

Changes the direction of a single bit from the I/O port.

digitalio→set_bitOpenDrain(**bitno**, **opendrain**)

Changes the electrical interface of a single bit from the I/O port.

digitalio→set_bitPolarity(**bitno**, **bitpolarity**)

Changes the polarity of a single bit from the I/O port.

digitalio→set_bitState(**bitno**, **bitstate**)

Sets a single bit of the I/O port.

digitalio→set_logicalName(**newval**)

Changes the logical name of the digital IO port.

digitalio→set_outputVoltage(**newval**)

Changes the voltage source used to drive output bits.

digitalio→set_portDirection(**newval**)

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→set_portOpenDrain(**newval**)

Changes the electrical interface for each bit of the port.

digitalio→set_portPolarity(**newval**)

Changes the polarity of all the bits of the port: For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

digitalio→set_portState(**newval**)

Changes the digital IO port state: bit 0 represents input 0, and so on.

digitalio→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

digitalio→toggle_bitState(**bitno**)

Reverts a single bit of the I/O port.

digitalio→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

digitalio→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDigitalIO.FindDigitalIO() YDigitalIO yFindDigitalIO()yFindDigitalIO()

Retrieves a digital IO port for a given identifier.

```
function FindDigitalIO( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the digital IO port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDigitalIO.isOnline()` to test if the digital IO port is indeed online at a given time. In case of ambiguity when looking for a digital IO port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the digital IO port

Returns :

a `YDigitalIO` object allowing you to drive the digital IO port.

YDigitalIO.FindDigitalIOInContext() YDigitalIO yFindDigitalIOInContext()

Retrieves a digital IO port for a given identifier in a YAPI context.

function **FindDigitalIOInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the digital IO port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDigitalIO.isOnline()` to test if the digital IO port is indeed online at a given time. In case of ambiguity when looking for a digital IO port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the digital IO port

Returns :

a `YDigitalIO` object allowing you to drive the digital IO port.

YDigitalIO.FirstDigitalIO() YDigitalIO yFirstDigitalIO()yFirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

function **FirstDigitalIO**( )

Use the method `YDigitalIO.nextDigitalIO()` to iterate on next digital IO ports.

Returns :

a pointer to a `YDigitalIO` object, corresponding to the first digital IO port currently online, or a `null` pointer if there are none.

YDigitalIO.FirstDigitalIOInContext() YDigitalIO
yFirstDigitalIOInContext()

Starts the enumeration of digital IO ports currently accessible.

function **FirstDigitalIOInContext**(**yctx**)

Use the method `YDigitalIO.nextDigitalIO()` to iterate on next digital IO ports.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDigitalIO` object, corresponding to the first digital IO port currently online, or a `null` pointer if there are none.

digitalio→clearCache()digitalio.clearCache() YDigitalIO

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the digital IO port attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

digitalio→delayedPulse()digitalio.delayedPulse() YDigitalIO

Schedules a pulse on a single bit for a specified duration.

function **delayedPulse**(**bitno**, **ms_delay**, **ms_duration**)

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno	the bit number; lowest bit has index 0
ms_delay	waiting time before the pulse, in milliseconds
ms_duration	desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→describe()digitalio.describe() YDigitalIO

Returns a short text that describes unambiguously the instance of the digital IO port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the digital IO port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

digitalio→get_advertisedValue() YDigitalIO
digitalio→advertisedValue()
digitalio.get_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the digital IO port (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

digitalio→get_bitDirection() YDigitalIO
digitalio→bitDirection()digitalio.get_bitDirection()

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

function **get_bitDirection**(**bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→get_bitOpenDrain() YDigitalIO
digitalio→bitOpenDrain()digitalio.get_bitOpenDrain()

Returns the type of electrical interface of a single bit from the I/O port.

function **get_bitOpenDrain**(**bitno**)

(0 means the bit is an input, 1 an output).

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

0 means the a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

On failure, throws an exception or returns a negative error code.

digitalio→get_bitPolarity() YDigitalIO
digitalio→bitPolarity()digitalio.get_bitPolarity()

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

function **get_bitPolarity**(**bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→get_bitState() YDigitalIO
digitalio→bitState()digitalio.get_bitState()

Returns the state of a single bit of the I/O port.

function **get_bitState**(**bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

the bit state (0 or 1)

On failure, throws an exception or returns a negative error code.

digitalio→get_errorMessage() YDigitalIO
digitalio→errorMessage()
digitalio.get_errorMessage()

Returns the error message of the latest error with the digital IO port.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the digital IO port object

digitalio→get_errorType() YDigitalIO
digitalio→errorType()digitalio.get_errorType()

Returns the numerical error code of the latest error with the digital IO port.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the digital IO port object

digitalio→get_friendlyName() YDigitalIO
digitalio→friendlyName()digitalio.get_friendlyName()

Returns a global identifier of the digital IO port in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the digital IO port if they are defined, otherwise the serial number of the module and the hardware identifier of the digital IO port (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the digital IO port using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

digitalio→get_functionDescriptor() YDigitalIO
digitalio→functionDescriptor()
digitalio.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

digitalio→get_functionId() YDigitalIO
digitalio→functionId()digitalio.get_functionId()

Returns the hardware identifier of the digital IO port, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the digital IO port (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

digitalio→get_hardwareId() YDigitalIO
digitalio→hardwareId()digitalio.get_hardwareId()

Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the digital IO port (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the digital IO port (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

digitalio→get_logicalName() YDigitalIO
digitalio→logicalName()digitalio.get_logicalName()

Returns the logical name of the digital IO port.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the digital IO port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

digitalio→get_module() YDigitalIO
digitalio→module()digitalio.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

digitalio→get_outputVoltage() YDigitalIO
digitalio→outputVoltage()
digitalio.get_outputVoltage()

Returns the voltage source used to drive output bits.

function **get_outputVoltage**( )

Returns :

a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

On failure, throws an exception or returns `Y_OUTPUTVOLTAGE_INVALID`.

digitalio→get_portDiags() YDigitalIO
digitalio→portDiags()digitalio.get_portDiags()

Returns the port state diagnostics (Yocto-IO and Yocto-MaxiIO-V2 only).

function **get_portDiags**( )

Bit 0 indicates a shortcut on output 0, etc. Bit 8 indicates a power failure, and bit 9 signals overheating (overcurrent). During normal use, all diagnostic bits should stay clear.

Returns :
an integer corresponding to the port state diagnostics (Yocto-IO and Yocto-MaxiIO-V2 only)

On failure, throws an exception or returns `Y_PORTDIAGS_INVALID`.

digitalio→get_portDirection() YDigitalIO
digitalio→portDirection()digitalio.get_portDirection()

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

function **get_portDirection**( )

Returns :

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

On failure, throws an exception or returns `Y_PORTDIRECTION_INVALID`.

digitalio→get_portOpenDrain() YDigitalIO
digitalio→portOpenDrain()
digitalio.get_portOpenDrain()

Returns the electrical interface for each bit of the port.

function **get_portOpenDrain**( )

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

Returns :
an integer corresponding to the electrical interface for each bit of the port

On failure, throws an exception or returns `Y_PORTOPENDRAIN_INVALID`.

digitalio→get_portPolarity() YDigitalIO
digitalio→portPolarity()digitalio.get_portPolarity()

Returns the polarity of all the bits of the port.

function **get_portPolarity**( )

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

Returns :
an integer corresponding to the polarity of all the bits of the port

On failure, throws an exception or returns `Y_PORTPOLARITY_INVALID`.

digitalio→get_portSize() YDigitalIO
digitalio→portSize()digitalio.get_portSize()

Returns the number of bits implemented in the I/O port.

function **get_portSize**( )

Returns :
an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns `Y_PORTSIZE_INVALID`.

digitalio→get_portState() YDigitalIO
digitalio→portState()digitalio.get_portState()

Returns the digital IO port state: bit 0 represents input 0, and so on.

function **get_portState**( )

Returns :
an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

On failure, throws an exception or returns `Y_PORTSTATE_INVALID`.

digitalio→get_userData() YDigitalIO
digitalio→userData()digitalio.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.

digitalio→isOnline()digitalio.isOnline() YDigitalIO

Checks if the digital IO port is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the digital IO port.

Returns :

`true` if the digital IO port can be reached, and `false` otherwise

digitalio→load()digitalio.load() YDigitalIO

Preloads the digital IO port cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→loadAttribute()digitalio.loadAttribute() YDigitalIO

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

digitalio→muteValueCallbacks() YDigitalIO
digitalio.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→nextDigitalIO()digitalio.nextDigitalIO() YDigitalIO

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

function **nextDigitalIO**( )

Returns :

a pointer to a `YDigitalIO` object, corresponding to a digital IO port currently online, or a `null` pointer if there are no more digital IO ports to enumerate.

digitalio→pulse()digitalio.pulse() YDigitalIO

Triggers a pulse on a single bit for a specified duration.

function **pulse**(**bitno**, **ms_duration**)

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno	the bit number; lowest bit has index 0
ms_duration	desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→registerValueCallback() YDigitalIO
digitalio.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.

digitalio→set_bitDirection() YDigitalIO
digitalio→setBitDirection()digitalio.set_bitDirection()

Changes the direction of a single bit from the I/O port.

function **set_bitDirection**(**bitno**, **bitdirection**)

Parameters :

bitno the bit number; lowest bit has index 0

bitdirection direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_bitOpenDrain() YDigitalIO
digitalio→setBitOpenDrain()
digitalio.set_bitOpenDrain()

Changes the electrical interface of a single bit from the I/O port.

function **set_bitOpenDrain**(**bitno**, **opendrain**)

Parameters :

bitno the bit number; lowest bit has index 0

opendrain 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_bitPolarity() YDigitalIO
digitalio→setBitPolarity()digitalio.set_bitPolarity()

Changes the polarity of a single bit from the I/O port.

function **set_bitPolarity**(**bitno**, **bitpolarity**)

Parameters :

bitno the bit number; lowest bit has index 0.

bitpolarity polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_bitState() YDigitalIO
digitalio→setBitState()digitalio.set_bitState()

Sets a single bit of the I/O port.

function **set_bitState**(**bitno**, **bitstate**)

Parameters :

bitno the bit number; lowest bit has index 0

bitstate the state of the bit (1 or 0)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_logicalName() YDigitalIO
digitalio→setLogicalName()
digitalio.set_logicalName()

Changes the logical name of the digital IO port.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the digital IO port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_outputVoltage() YDigitalIO
digitalio→setOutputVoltage()
digitalio.set_outputVoltage()

Changes the voltage source used to drive output bits.

function **set_outputVoltage**(**newval**)

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portDirection() YDigitalIO
digitalio→setPortDirection()
digitalio.set_portDirection()

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

function **set_portDirection**(**newval**)

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portOpenDrain() YDigitalIO
digitalio→setPortOpenDrain()
digitalio.set_portOpenDrain()

Changes the electrical interface for each bit of the port.

function **set_portOpenDrain**(**newval**)

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the electrical interface for each bit of the port

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portPolarity() YDigitalIO
digitalio→setPortPolarity()digitalio.set_portPolarity()

Changes the polarity of all the bits of the port: For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

function **set_portPolarity**(**newval**)

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

Parameters :

newval an integer corresponding to the polarity of all the bits of the port: For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portState() YDigitalIO
digitalio→setPortState()digitalio.set_portState()

Changes the digital IO port state: bit 0 represents input 0, and so on.

function **set_portState**(**newval**)

This function has no effect on bits configured as input in `portDirection`.

Parameters :

newval an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_userData() YDigitalIO
digitalio→setUserData()digitalio.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

digitalio→toggle_bitState()digitalio.toggle_bitState() YDigitalIO

Reverts a single bit of the I/O port.

function **toggle_bitState**(**bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→unmuteValueCallbacks() YDigitalIO
digitalio.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→wait_async()digitalio.wait_async() YDigitalIO

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.22. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_display.js'></script>
cpp	#include "yocto_display.h"
m	#import "yocto_display.h"
pas	uses yocto_display;
vb	yocto_display.vb
cs	yocto_display.cs
java	import com.yoctopuce.YoctoAPI.YDisplay;
uwp	import com.yoctopuce.YoctoAPI.YDisplay;
py	from yocto_display import *
php	require_once('yocto_display.php');
es	in HTML: <script src="../../lib/yocto_display.js"></script> in node.js: require('yoctolib-es2017/yocto_display.js');

Global functions
yFindDisplay(**func**) Retrieves a display for a given identifier.
yFindDisplayInContext(**yctx**, **func**) Retrieves a display for a given identifier in a YAPI context.
yFirstDisplay() Starts the enumeration of displays currently accessible.
yFirstDisplayInContext(**yctx**) Starts the enumeration of displays currently accessible.
`YDisplay` methods
display→clearCache() Invalidates the cache.
display→copyLayerContent(**srcLayerId**, **dstLayerId**) Copies the whole content of a layer to another layer.
display→describe() Returns a short text that describes unambiguously the instance of the display in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
display→fade(**brightness**, **duration**) Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.
display→get_advertisedValue() Returns the current value of the display (no more than 6 characters).
display→get_brightness() Returns the luminosity of the module informative leds (from 0 to 100).
display→get_displayHeight() Returns the display height, in pixels.
display→get_displayLayer(**layerId**) Returns a YDisplayLayer object that can be used to draw on the specified layer.
display→get_displayType()

Returns the display type: monochrome, gray levels or full color.

display→get_displayWidth()
Returns the display width, in pixels.

display→get_enabled()
Returns true if the screen is powered, false otherwise.

display→get_errorMessage()
Returns the error message of the latest error with the display.

display→get_errorType()
Returns the numerical error code of the latest error with the display.

display→get_friendlyName()
Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

display→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

display→get_functionId()
Returns the hardware identifier of the display, without reference to the module.

display→get_hardwareId()
Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

display→get_layerCount()
Returns the number of available layers to draw on.

display→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

display→get_logicalName()
Returns the logical name of the display.

display→get_module()
Gets the `YModule` object for the device on which the function is located.

display→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

display→get_orientation()
Returns the currently selected display orientation.

display→get_startupSeq()
Returns the name of the sequence to play when the displayed is powered on.

display→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

display→isOnline()
Checks if the display is currently reachable, without raising any error.

display→isOnline_async(**callback**, **context**)
Checks if the display is currently reachable, without raising any error (asynchronous version).

display→load(**msValidity**)
Preloads the display cache with a specified validity duration.

display→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

display→load_async(**msValidity**, **callback**, **context**)

Preloads the display cache with a specified validity duration (asynchronous version).

display→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

display→newSequence()

Starts to record all display commands into a sequence, for later replay.

display→nextDisplay()

Continues the enumeration of displays started using `yFirstDisplay()`.

display→pauseSequence(**delay_ms**)

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

display→playSequence(**sequenceName**)

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

display→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

display→resetAll()

Clears the display screen and resets all display layers to their default state.

display→saveSequence(**sequenceName**)

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

display→set_brightness(**newval**)

Changes the brightness of the display.

display→set_enabled(**newval**)

Changes the power state of the display.

display→set_logicalName(**newval**)

Changes the logical name of the display.

display→set_orientation(**newval**)

Changes the display orientation.

display→set_startupSeq(**newval**)

Changes the name of the sequence to play when the displayed is powered on.

display→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

display→stopSequence()

Stops immediately any ongoing sequence replay.

display→swapLayerContent(**layerIdA**, **layerIdB**)

Swaps the whole content of two layers.

display→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

display→upload(**pathname**, **content**)

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

display→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDisplay.FindDisplay() YDisplay yFindDisplay()yFindDisplay()

Retrieves a display for a given identifier.

function **FindDisplay**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the display

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FindDisplayInContext() YDisplay yFindDisplayInContext()

Retrieves a display for a given identifier in a YAPI context.

function **FindDisplayInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the display

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FirstDisplay() YDisplay yFirstDisplay()yFirstDisplay()

Starts the enumeration of displays currently accessible.

function **FirstDisplay**( )

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

Returns :

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

YDisplay.FirstDisplayInContext() yFirstDisplayInContext()

YDisplay

Starts the enumeration of displays currently accessible.

function **FirstDisplayInContext**(**yctx**)

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

display→clearCache()display.clearCache() YDisplay

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the display attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

display→copyLayerContent() YDisplay
display.copyLayerContent()

Copies the whole content of a layer to another layer.

function **copyLayerContent**(**srcLayerId**, **dstLayerId**)

The color and transparency of all the pixels from the destination layer are set to match the source pixels. This method only affects the displayed content, but does not change any property of the layer object. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

srcLayerId the identifier of the source layer (a number in range 0..layerCount-1)

dstLayerId the identifier of the destination layer (a number in range 0..layerCount-1)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→describe()display.describe() YDisplay

Returns a short text that describes unambiguously the instance of the display in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the display (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

display→fade()display.fade() YDisplay

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

function **fade**(**brightness**, **duration**)

Parameters :

brightness the new screen brightness

duration duration of the brightness transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→get_advertisedValue() YDisplay
display→advertisedValue()
display.get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the display (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

display→get_brightness() YDisplay
display→brightness()display.get_brightness()

Returns the luminosity of the module informative leds (from 0 to 100).

function **get_brightness**( )

Returns :
an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_BRIGHTNESS_INVALID`.

display→get_displayHeight() YDisplay
display→displayHeight()display.get_displayHeight()

Returns the display height, in pixels.

function **get_displayHeight**( )

Returns :
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns `Y_DISPLAYHEIGHT_INVALID`.

display→get_displayLayer() YDisplay
display→displayLayer()display.get_displayLayer()

Returns a YDisplayLayer object that can be used to draw on the specified layer.

function **get_displayLayer**(**layerId**)

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

Parameters :

layerId the identifier of the layer (a number in range 0..layerCount-1)

Returns :

an `YDisplayLayer` object

On failure, throws an exception or returns `null`.

display→get_displayType() YDisplay
display→displayType()display.get_displayType()

Returns the display type: monochrome, gray levels or full color.

function **get_displayType**( )

Returns :

a value among `Y_DISPLAYTYPE_MONO`, `Y_DISPLAYTYPE_GRAY` and `Y_DISPLAYTYPE_RGB` corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns `Y_DISPLAYTYPE_INVALID`.

display→get_displayWidth() YDisplay
display→displayWidth()display.get_displayWidth()

Returns the display width, in pixels.

function **get_displayWidth**( )

Returns :
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

display→get_enabled() YDisplay
display→enabled()display.get_enabled()

Returns true if the screen is powered, false otherwise.

function **get_enabled**( )

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

display→get_errorMessage() YDisplay
display→errorMessage()display.get_errorMessage()

Returns the error message of the latest error with the display.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the display object

display→get_errorType() YDisplay
display→errorType()display.get_errorType()

Returns the numerical error code of the latest error with the display.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the display object

display→get_friendlyName() YDisplay
display→friendlyName()display.get_friendlyName()

Returns a global identifier of the display in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the display if they are defined, otherwise the serial number of the module and the hardware identifier of the display (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the display using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

display→get_functionDescriptor() YDisplay
display→functionDescriptor()
display.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

display→get_functionId() YDisplay
display→functionId()display.get_functionId()

Returns the hardware identifier of the display, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the display (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

display→get_hardwareId() YDisplay
display→hardwareId()display.get_hardwareId()

Returns the unique hardware identifier of the display in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the display (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

display→get_layerCount() YDisplay
display→layerCount()display.get_layerCount()

Returns the number of available layers to draw on.

function **get_layerCount**( )

Returns :
an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns `Y_LAYERCOUNT_INVALID`.

display→get_layerHeight() YDisplay
display→layerHeight()display.get_layerHeight()

Returns the height of the layers to draw on, in pixels.

function **get_layerHeight**( )

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns `Y_LAYERHEIGHT_INVALID`.

display→get_layerWidth() YDisplay
display→layerWidth()display.get_layerWidth()

Returns the width of the layers to draw on, in pixels.

function **get_layerWidth**( )

Returns :
an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns `Y_LAYERWIDTH_INVALID`.

display→get_logicalName() YDisplay
display→logicalName()display.get_logicalName()

Returns the logical name of the display.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the display.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

display→get_module() YDisplay
display→module()display.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

display→get_orientation() YDisplay
display→orientation()display.get_orientation()

Returns the currently selected display orientation.

function **get_orientation**( )

Returns :

a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the currently selected display orientation

On failure, throws an exception or returns `Y_ORIENTATION_INVALID`.

display→get_startupSeq() YDisplay
display→startupSeq()display.get_startupSeq()

Returns the name of the sequence to play when the displayed is powered on.

function **get_startupSeq**( )

Returns :
a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns `Y_STARTUPSEQ_INVALID`.

display→get_userData() YDisplay
display→userData()display.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.

display→isOnline()display.isOnline() YDisplay

Checks if the display is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

Returns :
`true` if the display can be reached, and `false` otherwise

display→load()display.load() YDisplay

Preloads the display cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

display→loadAttribute()display.loadAttribute() YDisplay

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

display→muteValueCallbacks() YDisplay
display.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

display→newSequence()display.newSequence() YDisplay

Starts to record all display commands into a sequence, for later replay.

function **newSequence**( )

The name used to store the sequence is specified when calling `saveSequence()`, once the recording is complete.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→nextDisplay()display.nextDisplay() YDisplay

Continues the enumeration of displays started using `yFirstDisplay()`.

function **nextDisplay**( )

Returns :

a pointer to a `YDisplay` object, corresponding to a display currently online, or a `null` pointer if there are no more displays to enumerate.

display→pauseSequence()display.pauseSequence() YDisplay

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

function **pauseSequence**(**delay_ms**)

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

Parameters :

delay_ms the duration to wait, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→playSequence()display.playSequence() YDisplay

Replays a display sequence previously recorded using newSequence() and saveSequence().

function **playSequence**(**sequenceName**)

Parameters :

sequenceName the name of the newly created sequence

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→registerValueCallback() YDisplay
display.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.

display→resetAll()display.resetAll() YDisplay

Clears the display screen and resets all display layers to their default state.

function **resetAll**( )

Using this function in a sequence will kill the sequence play-back. Don't use that function to reset the display at sequence start-up.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→saveSequence()display.saveSequence() YDisplay

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

function **saveSequence**(**sequenceName**)

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_brightness() YDisplay
display→setBrightness()display.set_brightness()

Changes the brightness of the display.

function **set_brightness**(**newval**)

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the brightness of the display

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_enabled() YDisplay
display→setEnabled()display.set_enabled()

Changes the power state of the display.

function **set_enabled**(**newval**)

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the power state of the display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_logicalName() YDisplay
display→setLogicalName()display.set_logicalName()

Changes the logical name of the display.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the display.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_orientation() YDisplay
display→setOrientation()display.set_orientation()

Changes the display orientation.

function **set_orientation**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the display orientation

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_startupSeq() YDisplay
display→setStartupSeq()display.set_startupSeq()

Changes the name of the sequence to play when the displayed is powered on.

function **set_startupSeq**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the name of the sequence to play when the displayed is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_userData() YDisplay
display→setUserData()display.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

display→stopSequence()display.stopSequence() YDisplay

Stops immediately any ongoing sequence replay.

function **stopSequence**( )

The display is left as is.

Returns :
YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→swapLayerContent() YDisplay
display.swapLayerContent()

Swaps the whole content of two layers.

function **swapLayerContent**(**layerIdA**, **layerIdB**)

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between onae hidden layer and a visible layer, this method makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

layerIdA the first layer (a number in range 0..layerCount-1)

layerIdB the second layer (a number in range 0..layerCount-1)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→unmuteValueCallbacks() YDisplay
display.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

display→upload()display.upload() YDisplay

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

function **upload**(**pathname**, **content**)

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→wait_async()display.wait_async() YDisplay

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.23. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_display.js'></script>
cpp     #include "yocto_display.h"
m       #import "yocto_display.h"
pas     uses yocto_display;
vb      yocto_display.vb
cs      yocto_display.cs
java    import com.yoctopuce.YoctoAPI.YDisplay;
uwp     import com.yoctopuce.YoctoAPI.YDisplay;
py      from yocto_display import *
php     require_once('yocto_display.php');
es      in HTML: <script src="../../lib/yocto_display.js"></script>
        in node.js: require('yoctolib-es2017/yocto_display.js');
```

YDisplayLayer methods
displaylayer→clear() Erases the whole content of the layer (makes it fully transparent).
displaylayer→clearConsole() Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.
displaylayer→consoleOut(**text**) Outputs a message in the console area, and advances the console pointer accordingly.
displaylayer→drawBar(**x1**, **y1**, **x2**, **y2**) Draws a filled rectangular bar at a specified position.
displaylayer→drawBitmap(**x**, **y**, **w**, **bitmap**, **bgcol**) Draws a bitmap at the specified position.
displaylayer→drawCircle(**x**, **y**, **r**) Draws an empty circle at a specified position.
displaylayer→drawDisc(**x**, **y**, **r**) Draws a filled disc at a given position.
displaylayer→drawImage(**x**, **y**, **imagename**) Draws a GIF image at the specified position.
displaylayer→drawPixel(**x**, **y**) Draws a single pixel at the specified position.
displaylayer→drawRect(**x1**, **y1**, **x2**, **y2**) Draws an empty rectangle at a specified position.
displaylayer→drawText(**x**, **y**, **anchor**, **text**) Draws a text string at the specified position.
displaylayer→get_display() Gets parent YDisplay.
displaylayer→get_displayHeight() Returns the display height, in pixels.
displaylayer→get_displayWidth()

Returns the display width, in pixels.

displaylayer→get_layerHeight()

Returns the height of the layers to draw on, in pixels.

displaylayer→get_layerWidth()

Returns the width of the layers to draw on, in pixels.

displaylayer→hide()

Hides the layer.

displaylayer→lineTo(**x**, **y**)

Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(**x**, **y**)

Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()

Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(**color**)

Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

displaylayer→selectFont(**fontname**)

Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(**graylevel**)

Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(**mode**)

Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(**bgcol**)

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(**x1**, **y1**, **x2**, **y2**)

Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(**wordwrap**)

Sets up the wrapping behaviour used by the `consoleOut` function.

displaylayer→setLayerPosition(**x**, **y**, **scrollTime**)

Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()

Shows the layer.

displaylayer→clear()displaylayer.clear() YDisplayLayer

Erases the whole content of the layer (makes it fully transparent).

function **clear**( )

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset()` instead.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→clearConsole() YDisplayLayer
displaylayer.clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

function **clearConsole**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→consoleOut()displaylayer.consoleOut() YDisplayLayer

Outputs a message in the console area, and advances the console pointer accordingly.

function **consoleOut**(**text**)

The console pointer position is automatically moved to the beginning of the next line when a newline character is met, or when the right margin is hit. When the new text to display extends below the lower margin, the console area is automatically scrolled up.

Parameters :

text the message to display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBar()displaylayer.drawBar() YDisplayLayer

Draws a filled rectangular bar at a specified position.

function **drawBar**(**x1**, **y1**, **x2**, **y2**)

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBitmap() YDisplayLayer
displaylayer.drawBitmap()

Draws a bitmap at the specified position.

function **drawBitmap**(**x**, **y**, **w**, **bitmap**, **bgcol**)

The bitmap is provided as a binary object, where each pixel maps to a bit, from left to right and from top to bottom. The most significant bit of each byte maps to the leftmost pixel, and the least significant bit maps to the rightmost pixel. Bits set to 1 are drawn using the layer selected pen color. Bits set to 0 are drawn using the specified background gray level, unless -1 is specified, in which case they are not drawn at all (as if transparent).

Parameters :

x	the distance from left of layer to the left of the bitmap, in pixels
y	the distance from top of layer to the top of the bitmap, in pixels
w	the width of the bitmap, in pixels
bitmap	a binary object
bgcol	the background gray level to use for zero bits (0 = black, 255 = white), or -1 to leave the pixels unchanged

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawCircle()displaylayer.drawCircle() YDisplayLayer

Draws an empty circle at a specified position.

function **drawCircle**(**x**, **y**, **r**)

Parameters :

x the distance from left of layer to the center of the circle, in pixels

y the distance from top of layer to the center of the circle, in pixels

r the radius of the circle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawDisc()displaylayer.drawDisc() YDisplayLayer

Draws a filled disc at a given position.

function **drawDisc**(**x**, **y**, **r**)

Parameters :

x the distance from left of layer to the center of the disc, in pixels

y the distance from top of layer to the center of the disc, in pixels

r the radius of the disc, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawImage()displaylayer.drawImage() YDisplayLayer

Draws a GIF image at the specified position.

function **drawImage**(**x**, **y**, **imagename**)

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

Parameters :

x	the distance from left of layer to the left of the image, in pixels
y	the distance from top of layer to the top of the image, in pixels
imagename	the GIF file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawPixel()displaylayer.drawPixel() YDisplayLayer

Draws a single pixel at the specified position.

function **drawPixel**(**x**, **y**)

Parameters :

x the distance from left of layer, in pixels

y the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawRect()displaylayer.drawRect() YDisplayLayer

Draws an empty rectangle at a specified position.

function **drawRect**(**x1**, **y1**, **x2**, **y2**)

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawText()displaylayer.drawText() YDisplayLayer

Draws a text string at the specified position.

function **drawText**(**x**, **y**, **anchor**, **text**)

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

Parameters :

x	the distance from left of layer to the text anchor point, in pixels
y	the distance from top of layer to the text anchor point, in pixels
anchor	the text anchor point, chosen among the Y_ALIGN enumeration: Y_ALIGN_TOP_LEFT, Y_ALIGN_CENTER_LEFT, Y_ALIGN_BASELINE_LEFT, Y_ALIGN_BOTTOM_LEFT, Y_ALIGN_TOP_CENTER, Y_ALIGN_CENTER, Y_ALIGN_BASELINE_CENTER, Y_ALIGN_BOTTOM_CENTER, Y_ALIGN_TOP_DECIMAL, Y_ALIGN_CENTER_DECIMAL, Y_ALIGN_BASELINE_DECIMAL, Y_ALIGN_BOTTOM_DECIMAL, Y_ALIGN_TOP_RIGHT, Y_ALIGN_CENTER_RIGHT, Y_ALIGN_BASELINE_RIGHT, Y_ALIGN_BOTTOM_RIGHT.
text	the text string to draw

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→get_display() YDisplayLayer
displaylayer→display()displaylayer.get_display()

Gets parent YDisplay.

function **get_display**( )

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :
an `YDisplay` object

displaylayer→get_displayHeight() YDisplayLayer
displaylayer→displayHeight()
displaylayer.get_displayHeight()

Returns the display height, in pixels.

function **get_displayHeight**( )

Returns :
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth() YDisplayLayer
displaylayer→displayWidth()
displaylayer.get_displayWidth()

Returns the display width, in pixels.

function **get_displayWidth**( )

Returns :
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→get_layerHeight() YDisplayLayer
displaylayer→layerHeight()
displaylayer.get_layerHeight()

Returns the height of the layers to draw on, in pixels.

function **get_layerHeight**( )

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth() YDisplayLayer
displaylayer→layerWidth()
displaylayer.get_layerWidth()

Returns the width of the layers to draw on, in pixels.

function **get_layerWidth**( )

Returns :
an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERWIDTH_INVALID.

displaylayer→hide()displaylayer.hide() YDisplayLayer

Hides the layer.

function **hide**( )

The state of the layer is perserved but the layer is not displayed on the screen until the next call to `unhide()`. Hiding the layer can positively affect the drawing speed, since it postpones the rendering until all operations are completed (double-buffering).

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→lineTo()displaylayer.lineTo() YDisplayLayer

Draws a line from current drawing pointer position to the specified position.

function **lineTo**(**x**, **y**)

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

Parameters :

x the distance from left of layer to the end point of the line, in pixels

y the distance from top of layer to the end point of the line, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→moveTo()displaylayer.moveTo() YDisplayLayer

Moves the drawing pointer of this layer to the specified position.

function **moveTo**(**x**, **y**)

Parameters :

x the distance from left of layer, in pixels

y the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→reset()displaylayer.reset() YDisplayLayer

Reverts the layer to its initial state (fully transparent, default settings).

function **reset**( )

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectColorPen() YDisplayLayer
displaylayer.selectColorPen()

Selects the pen color for all subsequent drawing functions, including text drawing.

function **selectColorPen**(**color**)

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectEraser() YDisplayLayer
displaylayer.selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

```
function selectEraser( )
```

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectFont()displaylayer.selectFont() YDisplayLayer

Selects a font to use for the next text drawing functions, by providing the name of the font file.

function **selectFont**(**fontname**)

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

Parameters :

fontname the font file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectGrayPen() YDisplayLayer
displaylayer.selectGrayPen()

Selects the pen gray level for all subsequent drawing functions, including text drawing.

function **selectGrayPen**(**graylevel**)

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the lighest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

Parameters :

graylevel the desired gray level, from 0 to 255

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setAntialiasingMode() YDisplayLayer
displaylayer.setAntialiasingMode()

Enables or disables anti-aliasing for drawing oblique lines and circles.

function **setAntialiasingMode**(**mode**)

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

Parameters :

mode `true` to enable antialiasing, `false` to disable it.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleBackground() YDisplayLayer
displaylayer.setConsoleBackground()

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

function **setConsoleBackground**(**bgcol**)

Parameters :

bgcol the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleMargins() YDisplayLayer
displaylayer.setConsoleMargins()

Sets up display margins for the `consoleOut` function.

function **setConsoleMargins**(**x1**, **y1**, **x2**, **y2**)

Parameters :

x1 the distance from left of layer to the left margin, in pixels

y1 the distance from top of layer to the top margin, in pixels

x2 the distance from left of layer to the right margin, in pixels

y2 the distance from top of layer to the bottom margin, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleWordWrap() YDisplayLayer
displaylayer.setConsoleWordWrap()

Sets up the wrapping behaviour used by the `consoleOut` function.

function **setConsoleWordWrap**(**wordwrap**)

Parameters :

wordwrap `true` to wrap only between words, `false` to wrap on the last column anyway.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setLayerPosition() YDisplayLayer
displaylayer.setLayerPosition()

Sets the position of the layer relative to the display upper left corner.

function **setLayerPosition**(**x**, **y**, **scrollTime**)

When smooth scrolling is used, the display offset of the layer is automatically updated during the next milliseconds to animate the move of the layer.

Parameters :

x	the distance from left of display to the upper left corner of the layer
y	the distance from top of display to the upper left corner of the layer
scrollTime	number of milliseconds to use for smooth scrolling, or 0 if the scrolling should be immediate.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→unhide()displaylayer.unhide() YDisplayLayer

Shows the layer.

function **unhide**( )

Shows the layer again after a hide command.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.24. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_dualpower.js'></script>
cpp	#include "yocto_dualpower.h"
m	#import "yocto_dualpower.h"
pas	uses yocto_dualpower;
vb	yocto_dualpower.vb
cs	yocto_dualpower.cs
java	import com.yoctopuce.YoctoAPI.YDualPower;
uwp	import com.yoctopuce.YoctoAPI.YDualPower;
py	from yocto_dualpower import *
php	require_once('yocto_dualpower.php');
es	in HTML: <script src="../../lib/yocto_dualpower.js"></script> in node.js: require('yoctolib-es2017/yocto_dualpower.js');

Global functions
yFindDualPower(**func**) Retrieves a dual power control for a given identifier.
yFindDualPowerInContext(**yctx**, **func**) Retrieves a dual power control for a given identifier in a YAPI context.
yFirstDualPower() Starts the enumeration of dual power controls currently accessible.
yFirstDualPowerInContext(**yctx**) Starts the enumeration of dual power controls currently accessible.
`YDualPower` methods
dualpower→clearCache() Invalidates the cache.
dualpower→describe() Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
dualpower→get_advertisedValue() Returns the current value of the power control (no more than 6 characters).
dualpower→get_errorMessage() Returns the error message of the latest error with the power control.
dualpower→get_errorType() Returns the numerical error code of the latest error with the power control.
dualpower→get_extVoltage() Returns the measured voltage on the external power source, in millivolts.
dualpower→get_friendlyName() Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.
dualpower→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
dualpower→get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

dualpower→get_hardwareId()

Returns the unique hardware identifier of the power control in the form `SERIAL.FUNCTIONID`.

dualpower→get_logicalName()

Returns the logical name of the power control.

dualpower→get_module()

Gets the `YModule` object for the device on which the function is located.

dualpower→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

dualpower→get_powerControl()

Returns the selected power source for module functions that require lots of current.

dualpower→get_powerState()

Returns the current power source for module functions that require lots of current.

dualpower→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

dualpower→isOnline()

Checks if the power control is currently reachable, without raising any error.

dualpower→isOnline_async(**callback**, **context**)

Checks if the power control is currently reachable, without raising any error (asynchronous version).

dualpower→load(**msValidity**)

Preloads the power control cache with a specified validity duration.

dualpower→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

dualpower→load_async(**msValidity**, **callback**, **context**)

Preloads the power control cache with a specified validity duration (asynchronous version).

dualpower→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

dualpower→nextDualPower()

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

dualpower→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

dualpower→set_logicalName(**newval**)

Changes the logical name of the power control.

dualpower→set_powerControl(**newval**)

Changes the selected power source for module functions that require lots of current.

dualpower→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

dualpower→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

dualpower→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDualPower.FindDualPower() YDualPower yFindDualPower()yFindDualPower()

Retrieves a dual power control for a given identifier.

function **FindDualPower**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDualPower.isOnline()` to test if the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

YDualPower.FindDualPowerInContext() YDualPower yFindDualPowerInContext()

Retrieves a dual power control for a given identifier in a YAPI context.

function **FindDualPowerInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

YDualPower.FirstDualPower() YDualPower yFirstDualPower()yFirstDualPower()

Starts the enumeration of dual power controls currently accessible.

function **FirstDualPower**( )

Use the method `YDualPower.nextDualPower()` to iterate on next dual power controls.

Returns :

a pointer to a `YDualPower` object, corresponding to the first dual power control currently online, or a `null` pointer if there are none.

YDualPower.FirstDualPowerInContext() YDualPower yFirstDualPowerInContext()

Starts the enumeration of dual power controls currently accessible.

function **FirstDualPowerInContext**(**yctx**)

Use the method `YDualPower.nextDualPower()` to iterate on next dual power controls.

Parameters :

yctx a YAPI context.

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→clearCache()dualpower.clearCache() YDualPower

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the power control attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

dualpower→describe()dualpower.describe() YDualPower

Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the power control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

dualpower→get_advertisedValue() YDualPower
dualpower→advertisedValue()
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() YDualPower
dualpower→errorMessage()
dualpower.get_errorMessage()

Returns the error message of the latest error with the power control.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the power control object

dualpower→get_errorType() YDualPower
dualpower→errorType()dualpower.get_errorType()

Returns the numerical error code of the latest error with the power control.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the power control object

dualpower→get_extVoltage() YDualPower
dualpower→extVoltage()dualpower.get_extVoltage()

Returns the measured voltage on the external power source, in millivolts.

function **get_extVoltage**( )

Returns :
an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns `Y_EXTVOLTAGE_INVALID`.

dualpower→get_friendlyName() YDualPower
dualpower→friendlyName()
dualpower.get_friendlyName()

Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the power control if they are defined, otherwise the serial number of the module and the hardware identifier of the power control (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the power control using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

dualpower→get_functionDescriptor() YDualPower
dualpower→functionDescriptor()
dualpower.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

dualpower→get_functionId() YDualPower
dualpower→functionId()dualpower.get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the power control (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

dualpower→get_hardwareId() YDualPower
dualpower→hardwareId()dualpower.get_hardwareId()

Returns the unique hardware identifier of the power control in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the power control (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the power control (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

dualpower→get_logicalName() YDualPower
dualpower→logicalName()
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() YDualPower
dualpower→module()dualpower.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

dualpower→get_powerControl() YDualPower
dualpower→powerControl()
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() YDualPower
dualpower→powerState()
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() YDualPower
dualpower→userData()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()dualpower.isOnline() YDualPower

Checks if the power control is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power control.

Returns :

`true` if the power control can be reached, and `false` otherwise

dualpower→load()dualpower.load() YDualPower

Preloads the power control cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()dualpower.loadAttribute() YDualPower

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

dualpower→muteValueCallbacks() YDualPower
dualpower.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

dualpower→nextDualPower() YDualPower
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() YDualPower
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() YDualPower
dualpower→setLogicalName()
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() YDualPower
dualpower→setPowerControl()
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() YDualPower
dualpower→setUserData()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

dualpower→unmuteValueCallbacks() YDualPower
dualpower.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

dualpower→wait_async()dualpower.wait_async() YDualPower

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.25. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_files.js'></script>
cpp	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
uwp	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *
php	require_once('yocto_files.php');
es	in HTML: <script src="../../lib/yocto_files.js"></script> in node.js: require('yoctolib-es2017/yocto_files.js');

Global functions

yFindFiles(**func**)

Retrieves a filesystem for a given identifier.

yFindFilesInContext(**yctx**, **func**)

Retrieves a filesystem for a given identifier in a YAPI context.

yFirstFiles()

Starts the enumeration of filesystems currently accessible.

yFirstFilesInContext(**yctx**)

Starts the enumeration of filesystems currently accessible.

YFiles methods

files→clearCache()

Invalidates the cache.

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

files→download(**pathname**)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(**pathname**, **callback**, **context**)

Downloads the requested file and returns a binary buffer with its content.

files→fileExist(**filename**)

Test if a file exist on the filesystem of the module.

files→format_fs()

Reinitialize the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()
Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()
Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()
Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

files→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

files→get_functionId()
Returns the hardware identifier of the filesystem, without reference to the module.

files→get_hardwareId()
Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

files→get_list(**pattern**)
Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

files→get_logicalName()
Returns the logical name of the filesystem.

files→get_module()
Gets the `YModule` object for the device on which the function is located.

files→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

files→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

files→isOnline()
Checks if the filesystem is currently reachable, without raising any error.

files→isOnline_async(**callback**, **context**)
Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

files→load(**msValidity**)
Preloads the filesystem cache with a specified validity duration.

files→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

files→load_async(**msValidity**, **callback**, **context**)
Preloads the filesystem cache with a specified validity duration (asynchronous version).

files→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

files→nextFiles()
Continues the enumeration of filesystems started using `yFirstFiles()`.

files→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

files→remove(**pathname**)
Deletes a file, given by its full path name, from the filesystem.

files→set_logicalName(**newval**)
Changes the logical name of the filesystem.

files→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

files→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

files→upload(**pathname**, **content**)

Uploads a file to the filesystem, to the specified full path name.

files→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YFiles.FindFiles() YFiles
yFindFiles()yFindFiles()

Retrieves a filesystem for a given identifier.

```
function FindFiles( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the filesystem

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FindFilesInContext() YFiles
yFindFilesInContext()

Retrieves a filesystem for a given identifier in a YAPI context.

function **FindFilesInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the filesystem

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles() YFiles yFirstFiles()yFirstFiles()

Starts the enumeration of filesystems currently accessible.

function **FirstFiles**( )

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

YFiles.FirstFilesInContext() yFirstFilesInContext()

YFiles

Starts the enumeration of filesystems currently accessible.

function **FirstFilesInContext**(**yctx**)

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

files→clearCache()files.clearCache() YFiles

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the filesystem attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

files→describe()files.describe() YFiles

Returns a short text that describes unambiguously the instance of the filesystem in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the filesystem (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

files→download()files.download() YFiles

Downloads the requested file and returns a binary buffer with its content.

function **download**(**pathname**)

Parameters :

pathname path and name of the file to download

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

files→fileExist()files.fileExist() YFiles

Test if a file exist on the filesystem of the module.

function **fileExist**(**filename**)

Parameters :

filename the file name to test.

Returns :

a true if the file existe, false ortherwise.

On failure, throws an exception.

files→format_fs()files.format_fs() YFiles

Reinitialize the filesystem to its clean, unfragmented, empty state.

function **format_fs**( )

All files previously uploaded are permanently lost.

Returns :
YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→get_advertisedValue() YFiles
files→advertisedValue()files.get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the filesystem (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

files→get_errorMessage() YFiles
files→errorMessage()files.get_errorMessage()

Returns the error message of the latest error with the filesystem.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the filesystem object

files→get_errorType() YFiles
files→errorType()files.get_errorType()

Returns the numerical error code of the latest error with the filesystem.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the filesystem object

files→get_filesCount() YFiles
files→filesCount()files.get_filesCount()

Returns the number of files currently loaded in the filesystem.

function **get_filesCount**( )

Returns :
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `Y_FILESCOUNT_INVALID`.

files→get_freeSpace() YFiles
files→freeSpace()files.get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

function **get_freeSpace**( )

Returns :
an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns Y_FREESPACE_INVALID.

files→get_friendlyName() YFiles
files→friendlyName()files.get_friendlyName()

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

files→get_functionDescriptor() YFiles
files→functionDescriptor()
files.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

files→get_functionId() YFiles
files→functionId()files.get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the filesystem (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

files→get_hardwareId() YFiles
files→hardwareId()files.get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the filesystem (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

files→get_list() YFiles
files→list()files.get_list()

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

function **get_list**(**pattern**)

Parameters :

pattern an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

Returns :

a list of `YFileRecord` objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName() YFiles
files→logicalName()files.get_logicalName()

Returns the logical name of the filesystem.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the filesystem.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

files→get_module() YFiles
files→module()files.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

files→get_userData() YFiles
files→userData()files.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.

files→isOnline()files.isOnline() YFiles

Checks if the filesystem is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

Returns :

`true` if the filesystem can be reached, and `false` otherwise

files→load()files.load() YFiles

Preloads the filesystem cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→loadAttribute()files.loadAttribute() YFiles

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

files→muteValueCallbacks() YFiles
files.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→nextFiles()files.nextFiles() YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

function **nextFiles**( )

Returns :

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

files→registerValueCallback() YFiles
files.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.

files→remove()files.remove() YFiles

Deletes a file, given by its full path name, from the filesystem.

function **remove**(**pathname**)

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_logicalName() YFiles
files→setLogicalName()files.set_logicalName()

Changes the logical name of the filesystem.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the filesystem.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_userData() YFiles
files→setUserData()files.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

files→unmuteValueCallbacks() YFiles
files.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→upload()files.upload() YFiles

Uploads a file to the filesystem, to the specified full path name.

function **upload**(**pathname**, **content**)

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→wait_async()files.wait_async() YFiles

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.26. Control interface for the firmware update process

The YFirmwareUpdate class let you control the firmware update of a Yoctopuce module. This class should not be instantiate directly, instead the method `updateFirmware` should be called to get an instance of YFirmwareUpdate.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');

Global functions
yCheckFirmware(**serial**, **path**, **minrelease**)
Test if the byn file is valid for this module.
yGetAllBootLoaders()
Returns a list of all the modules in "firmware update" mode.
yGetAllBootLoadersInContext(**yctx**)
Returns a list of all the modules in "firmware update" mode.
`YFirmwareUpdate` methods
firmwareupdate→get_progress()
Returns the progress of the firmware update, on a scale from 0 to 100.
firmwareupdate→get_progressMessage()
Returns the last progress message of the firmware update process.
firmwareupdate→startUpdate()
Starts the firmware update process.

YFirmwareUpdate.CheckFirmware() yCheckFirmware()yCheckFirmware() — YFirmwareUpdate

Test if the byn file is valid for this module.

function **CheckFirmware**(**serial**, **path**, **minrelease**)

It is possible to pass a directory instead of a file. In that case, this method returns the path of the most recent appropriate byn file. This method will ignore any firmware older than minrelease.

Parameters :

serial	the serial number of the module to update
path	the path of a byn file or a directory that contains byn files
minrelease	a positive integer

Returns :

: the path of the byn file to use, or an empty string if no byn files matches the requirement

On failure, returns a string that starts with "error:".

YFirmwareUpdate.GetAllBootLoaders() YFirmwareUpdate yGetAllBootLoaders()

Returns a list of all the modules in "firmware update" mode.

function **GetAllBootLoaders**( )

Only devices connected over USB are listed. For devices connected to a YoctoHub, you must connect yourself to the YoctoHub web interface.

Returns :
an array of strings containing the serial numbers of devices in "firmware update" mode.

YFirmwareUpdate.GetAllBootLoadersInContext() YFirmwareUpdate yGetAllBootLoadersInContext()

Returns a list of all the modules in "firmware update" mode.

function **GetAllBootLoadersInContext**(**yctx**)

Only devices connected over USB are listed. For devices connected to a YoctoHub, you must connect to the YoctoHub web interface.

Parameters :

yctx a YAPI context.

Returns :

an array of strings containing the serial numbers of devices in "firmware update" mode.

firmwareupdate→get_progress() YFirmwareUpdate
firmwareupdate→progress()
firmwareupdate.get_progress()

Returns the progress of the firmware update, on a scale from 0 to 100.

function **get_progress**( )

When the object is instantiated, the progress is zero. The value is updated during the firmware update process until the value of 100 is reached. The 100 value means that the firmware update was completed successfully. If an error occurs during the firmware update, a negative value is returned, and the error message can be retrieved with `get_progressMessage`.

Returns :
an integer in the range 0 to 100 (percentage of completion) or a negative error code in case of failure.

firmwareupdate→get_progressMessage() YFirmwareUpdate
firmwareupdate→progressMessage()
firmwareupdate.get_progressMessage()

Returns the last progress message of the firmware update process.

function **get_progressMessage**( )

If an error occurs during the firmware update process, the error message is returned

Returns :
a string with the latest progress message, or the error message.

firmwareupdate→startUpdate() YFirmwareUpdate
firmwareupdate.startUpdate()

Starts the firmware update process.

function **startUpdate**( )

This method starts the firmware update process in background. This method returns immediately. You can monitor the progress of the firmware update with the `get_progress()` and `get_progressMessage()` methods.

Returns :
an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure returns a negative error code.

3.27. GenericSensor function interface

The YGenericSensor class allows you to read and configure Yoctopuce signal transducers. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to configure the automatic conversion between the measured signal and the corresponding engineering unit.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_genericsensor.js'></script>
cpp     #include "yocto_genericsensor.h"
m       #import "yocto_genericsensor.h"
pas     uses yocto_genericsensor;
vb      yocto_genericsensor.vb
cs      yocto_genericsensor.cs
java    import com.yoctopuce.YoctoAPI.YGenericSensor;
uwp     import com.yoctopuce.YoctoAPI.YGenericSensor;
py      from yocto_genericsensor import *
php     require_once('yocto_genericsensor.php');
es      in HTML: <script src="../../lib/yocto_genericsensor.js"></script>
        in node.js: require('yoctolib-es2017/yocto_genericsensor.js');
```

Global functions
yFindGenericSensor(**func**) Retrieves a generic sensor for a given identifier.
yFindGenericSensorInContext(**yctx**, **func**) Retrieves a generic sensor for a given identifier in a YAPI context.
yFirstGenericSensor() Starts the enumeration of generic sensors currently accessible.
yFirstGenericSensorInContext(**yctx**) Starts the enumeration of generic sensors currently accessible.
`YGenericSensor` methods
genericsensor→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
genericsensor→clearCache() Invalidates the cache.
genericsensor→describe() Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
genericsensor→get_advertisedValue() Returns the current value of the generic sensor (no more than 6 characters).
genericsensor→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
genericsensor→get_currentValue() Returns the current measured value.
genericsensor→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
genericsensor→get_errorMessage() Returns the error message of the latest error with the generic sensor.

genericsensor→get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

genericsensor→get_friendlyName()

Returns a global identifier of the generic sensor in the format MODULE_NAME.FUNCTION_NAME.

genericsensor→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

genericsensor→get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

genericsensor→get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form SERIAL.FUNCTIONID.

genericsensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

genericsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

genericsensor→get_logicalName()

Returns the logical name of the generic sensor.

genericsensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

genericsensor→get_module()

Gets the YModule object for the device on which the function is located.

genericsensor→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

genericsensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

genericsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

genericsensor→get_resolution()

Returns the resolution of the measured values.

genericsensor→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

genericsensor→get_signalBias()

Returns the electric signal bias for zero shift adjustment.

genericsensor→get_signalRange()

Returns the electric signal range used by the sensor.

genericsensor→get_signalSampling()

Returns the electric signal sampling method to use.

genericsensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericsensor→get_signalValue()

Returns the current value of the electrical signal measured by the sensor.

genericsensor→get_unit()

Returns the measuring unit for the measure.

genericsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

genericsensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericsensor→isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

genericsensor→isOnline_async(**callback**, **context**)

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

genericsensor→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

genericsensor→load(**msValidity**)

Preloads the generic sensor cache with a specified validity duration.

genericsensor→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

genericsensor→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

genericsensor→load_async(**msValidity**, **callback**, **context**)

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

genericsensor→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

genericsensor→nextGenericSensor()

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

genericsensor→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

genericsensor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

genericsensor→set_highestValue(**newval**)

Changes the recorded maximal value observed.

genericsensor→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

genericsensor→set_logicalName(**newval**)

Changes the logical name of the generic sensor.

genericsensor→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

genericsensor→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

genericsensor→set_resolution(**newval**)

Changes the resolution of the measured physical values.

genericsensor→set_signalBias(**newval**)

Changes the electric signal bias for zero shift adjustment.

genericsensor→set_signalRange(**newval**)

Changes the electric signal range used by the sensor.

genericsensor→set_signalSampling(**newval**)

Changes the electric signal sampling method to use.

genericsensor→set_unit(**newval**)

Changes the measuring unit for the measured value.

genericsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

genericsensor→set_valueRange(newval)

Changes the physical value range measured by the sensor.

genericsensor→startDataLogger()

Starts the data logger on the device.

genericsensor→stopDataLogger()

Stops the datalogger on the device.

genericsensor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

genericsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

genericsensor→zeroAdjust()

Adjusts the signal bias so that the current signal value is need precisely as zero.

YGenericSensor.FindGenericSensor() YGenericSensor yFindGenericSensor()yFindGenericSensor()

Retrieves a generic sensor for a given identifier.

function **FindGenericSensor**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the generic sensor

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FindGenericSensorInContext() YGenericSensor yFindGenericSensorInContext()

Retrieves a generic sensor for a given identifier in a YAPI context.

function **FindGenericSensorInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the generic sensor

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FirstGenericSensor() YGenericSensor yFirstGenericSensor()yFirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

function **FirstGenericSensor**( )

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

Returns :

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a `null` pointer if there are none.

YGenericSensor.FirstGenericSensorInContext() YGenericSensor yFirstGenericSensorInContext()

Starts the enumeration of generic sensors currently accessible.

function **FirstGenericSensorInContext**(**yctx**)

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a `null` pointer if there are none.

genericsensor→calibrateFromPoints() YGenericSensor
genericsensor.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→clearCache() YGenericSensor
genericsensor.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the generic sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

genericsensor→describe()genericsensor.describe() YGenericSensor

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the generic sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

genericsensor→get_advertisedValue() YGenericSensor
genericsensor→advertisedValue()
genericsensor.get_advertisedValue()

Returns the current value of the generic sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the generic sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

genericsensor→get_currentRawValue() YGenericSensor
genericsensor→currentRawValue()
genericsensor.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`.

genericsensor→get_currentValue() YGenericSensor
genericsensor→currentValue()
genericsensor.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`.

genericsensor→get_dataLogger() YGenericSensor
genericsensor→dataLogger()
genericsensor.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

genericsensor→get_errorMessage() YGenericSensor
genericsensor→errorMessage()
genericsensor.get_errorMessage()

Returns the error message of the latest error with the generic sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the generic sensor object

genericsensor→get_errorType() YGenericSensor
genericsensor→errorType()
genericsensor.get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the generic sensor object

genericsensor→get_friendlyName() YGenericSensor
genericsensor→friendlyName()
genericsensor.get_friendlyName()

Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the generic sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the generic sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the generic sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

genericsensor→get_functionDescriptor() YGenericSensor
genericsensor→functionDescriptor()
genericsensor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

genericsensor→get_functionId() YGenericSensor
genericsensor→functionId()
genericsensor.get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the generic sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

genericsensor→get_hardwareId() YGenericSensor
genericsensor→hardwareId()
genericsensor.get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the generic sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the generic sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

genericsensor→get_highestValue() YGenericSensor
genericsensor→highestValue()
genericsensor.get_highestValue()

Returns the maximal value observed for the measure since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

genericsensor→get_logFrequency() YGenericSensor
genericsensor→logFrequency()
genericsensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

genericsensor→get_logicalName() YGenericSensor
genericsensor→logicalName()
genericsensor.get_logicalName()

Returns the logical name of the generic sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the generic sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

genericsensor→get_lowestValue() YGenericSensor
genericsensor→lowestValue()
genericsensor.get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

genericsensor→get_module() YGenericSensor
genericsensor→module()
genericsensor.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

genericsensor→get_recordedData() YGenericSensor
genericsensor→recordedData()
genericsensor.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

genericsensor→get_reportFrequency() YGenericSensor
genericsensor→reportFrequency()
genericsensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

genericsensor→get_resolution() YGenericSensor
genericsensor→resolution()
genericsensor.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

genericsensor→get_sensorState() YGenericSensor
genericsensor→sensorState()
genericsensor.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

genericsensor→get_signalBias() YGenericSensor
genericsensor→signalBias()
genericsensor.get_signalBias()

Returns the electric signal bias for zero shift adjustment.

function **get_signalBias**( )

A positive bias means that the signal is over-reporting the measure, while a negative bias means that the signal is underreporting the measure.

Returns :
a floating point number corresponding to the electric signal bias for zero shift adjustment

On failure, throws an exception or returns `Y_SIGNALBIAS_INVALID`.

genericsensor→get_signalRange() YGenericSensor
genericsensor→signalRange()
genericsensor.get_signalRange()

Returns the electric signal range used by the sensor.

function **get_signalRange**( )

Returns :
a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns `Y_SIGNALRANGE_INVALID`.

genericsensor→get_signalSampling() YGenericSensor
genericsensor→signalSampling()
genericsensor.get_signalSampling()

Returns the electric signal sampling method to use.

function **get_signalSampling**( )

The HIGH_RATE method uses the highest sampling frequency, without any filtering. The HIGH_RATE_FILTERED method adds a windowed 7-sample median filter. The LOW_NOISE method uses a reduced acquisition frequency to reduce noise. The LOW_NOISE_FILTERED method combines a reduced frequency with the median filter to get measures as stable as possible when working on a noisy signal.

Returns :

a value among Y_SIGNALSAMPLING_HIGH_RATE, Y_SIGNALSAMPLING_HIGH_RATE_FILTERED, Y_SIGNALSAMPLING_LOW_NOISE and Y_SIGNALSAMPLING_LOW_NOISE_FILTERED corresponding to the electric signal sampling method to use

On failure, throws an exception or returns Y_SIGNALSAMPLING_INVALID.

genericsensor→get_signalUnit() YGenericSensor
genericsensor→signalUnit()
genericsensor.get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

function **get_signalUnit**( )

Returns :
a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns `Y_SIGNALUNIT_INVALID`.

genericsensor→get_signalValue() YGenericSensor
genericsensor→signalValue()
genericsensor.get_signalValue()

Returns the current value of the electrical signal measured by the sensor.

function **get_signalValue**( )

Returns :
a floating point number corresponding to the current value of the electrical signal measured by the sensor

On failure, throws an exception or returns `Y_SIGNALVALUE_INVALID`.

genericsensor→get_unit() YGenericSensor
genericsensor→unit()genericsensor.get_unit()

Returns the measuring unit for the measure.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

genericsensor→get_userData() YGenericSensor
genericsensor→userData()
genericsensor.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.

genericsensor→get_valueRange() YGenericSensor
genericsensor→valueRange()
genericsensor.get_valueRange()

Returns the physical value range measured by the sensor.

function **get_valueRange**( )

Returns :
a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns `Y_VALUERANGE_INVALID`.

genericsensor→isOnline()genericsensor.isOnline() YGenericSensor

Checks if the generic sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the generic sensor.

Returns :

`true` if the generic sensor can be reached, and `false` otherwise

genericsensor→load()genericsensor.load() YGenericSensor

Preloads the generic sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→loadAttribute() YGenericSensor
genericsensor.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

genericsensor→loadCalibrationPoints() YGenericSensor
genericsensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→muteValueCallbacks() YGenericSensor
genericsensor.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→nextGenericSensor() YGenericSensor
genericsensor.nextGenericSensor()

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

function **nextGenericSensor**( )

Returns :

a pointer to a `YGenericSensor` object, corresponding to a generic sensor currently online, or a `null` pointer if there are no more generic sensors to enumerate.

genericsensor→registerTimedReportCallback() YGenericSensor
genericsensor.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

genericsensor→registerValueCallback() YGenericSensor
genericsensor.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.

genericsensor→set_highestValue() YGenericSensor
genericsensor→setHighestValue()
genericsensor.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.

genericsensor→set_logFrequency() YGenericSensor
genericsensor→setLogFrequency()
genericsensor.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logicalName() YGenericSensor
genericsensor→setLogicalName()
genericsensor.set_logicalName()

Changes the logical name of the generic sensor.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the generic sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_lowestValue() YGenericSensor
genericsensor→setLowestValue()
genericsensor.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.

genericsensor→set_reportFrequency() YGenericSensor
genericsensor→setReportFrequency()
genericsensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_resolution() YGenericSensor
genericsensor→setResolution()
genericsensor.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalBias() YGenericSensor
genericsensor→setSignalBias()
genericsensor.set_signalBias()

Changes the electric signal bias for zero shift adjustment.

function **set_signalBias**(**newval**)

If your electric signal reads positif when it should be zero, setup a positive signalBias of the same value to fix the zero shift.

Parameters :

newval a floating point number corresponding to the electric signal bias for zero shift adjustment

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalRange() YGenericSensor
genericsensor→setSignalRange()
genericsensor.set_signalRange()

Changes the electric signal range used by the sensor.

function **set_signalRange**(**newval**)

Default value is "-999999.999...999999.999".

Parameters :

newval a string corresponding to the electric signal range used by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalSampling() YGenericSensor
genericsensor→setSignalSampling()
genericsensor.set_signalSampling()

Changes the electric signal sampling method to use.

function **set_signalSampling**(**newval**)

The HIGH_RATE method uses the highest sampling frequency, without any filtering. The HIGH_RATE_FILTERED method adds a windowed 7-sample median filter. The LOW_NOISE method uses a reduced acquisition frequency to reduce noise. The LOW_NOISE_FILTERED method combines a reduced frequency with the median filter to get measures as stable as possible when working on a noisy signal.

Parameters :

newval a value among Y_SIGNALSAMPLING_HIGH_RATE, Y_SIGNALSAMPLING_HIGH_RATE_FILTERED, Y_SIGNALSAMPLING_LOW_NOISE and Y_SIGNALSAMPLING_LOW_NOISE_FILTERED corresponding to the electric signal sampling method to use

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_unit() YGenericSensor
genericsensor→setUnit()genericsensor.set_unit()

Changes the measuring unit for the measured value.

```
function set_unit( newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the measuring unit for the measured value

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_userData() YGenericSensor
genericsensor→setUserData()
genericsensor.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

genericsensor→set_valueRange() YGenericSensor
genericsensor→setValueRange()
genericsensor.set_valueRange()

Changes the physical value range measured by the sensor.

```
function set_valueRange( newval)
```

As a side effect, the range modification may automatically modify the display resolution. Default value is "-999999.999...999999.999".

Parameters :

newval a string corresponding to the physical value range measured by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→startDataLogger() YGenericSensor
genericsensor.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

genericsensor→stopDataLogger() YGenericSensor
genericsensor.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

genericsensor→unmuteValueCallbacks() YGenericSensor
genericsensor.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→wait_async() YGenericSensor
genericsensor.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

genericsensor→zeroAdjust() YGenericSensor
genericsensor.zeroAdjust()

Adjusts the signal bias so that the current signal value is need precisely as zero.

function **zeroAdjust**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.28. GPS function interface

The Gps function allows you to extract positionning data from the GPS device. This class can provides complete positionning information: However, if you whish to define callbacks on position changes, you should use the YLatitude et YLongitude classes.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gps.js'></script>
cpp	#include "yocto_gps.h"
m	#import "yocto_gps.h"
pas	uses yocto_gps;
vb	yocto_gps.vb
cs	yocto_gps.cs
java	import com.yoctopuce.YoctoAPI.YGps;
uwp	import com.yoctopuce.YoctoAPI.YGps;
py	from yocto_gps import *
php	require_once('yocto_gps.php');
es	in HTML: <script src="../../lib/yocto_gps.js"></script> in node.js: require('yoctolib-es2017/yocto_gps.js');

Global functions
yFindGps(**func**) Retrieves a GPS for a given identifier.
yFindGpsInContext(**yctx**, **func**) Retrieves a GPS for a given identifier in a YAPI context.
yFirstGps() Starts the enumeration of GPS currently accessible.
yFirstGpsInContext(**yctx**) Starts the enumeration of GPS currently accessible.
YGps methods
gps→clearCache() Invalidates the cache.
gps→describe() Returns a short text that describes unambiguously the instance of the GPS in the form TYPE(NAME)=SERIAL.FUNCTIONID.
gps→get_advertisedValue() Returns the current value of the GPS (no more than 6 characters).
gps→get_altitude() Returns the current altitude.
gps→get_coordSystem() Returns the representation system used for positioning data.
gps→get_dateTime() Returns the current time in the form "YYYY/MM/DD hh:mm:ss".
gps→get_dilution() Returns the current horizontal dilution of precision, the smaller that number is, the better .
gps→get_direction() Returns the current move bearing in degrees, zero is the true (geographic) north.
gps→get_errorMessage()

Returns the error message of the latest error with the GPS.

gps→get_errorType()
Returns the numerical error code of the latest error with the GPS.

gps→get_friendlyName()
Returns a global identifier of the GPS in the format `MODULE_NAME.FUNCTION_NAME`.

gps→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

gps→get_functionId()
Returns the hardware identifier of the GPS, without reference to the module.

gps→get_groundSpeed()
Returns the current ground speed in Km/h.

gps→get_hardwareId()
Returns the unique hardware identifier of the GPS in the form `SERIAL.FUNCTIONID`.

gps→get_isFixed()
Returns TRUE if the receiver has found enough satellites to work.

gps→get_latitude()
Returns the current latitude.

gps→get_logicalName()
Returns the logical name of the GPS.

gps→get_longitude()
Returns the current longitude.

gps→get_module()
Gets the `YModule` object for the device on which the function is located.

gps→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

gps→get_satCount()
Returns the count of visible satellites.

gps→get_unixTime()
Returns the current time in Unix format (number of seconds elapsed since Jan 1st, 1970).

gps→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

gps→get_utcOffset()
Returns the number of seconds between current time and UTC time (time zone).

gps→isOnline()
Checks if the GPS is currently reachable, without raising any error.

gps→isOnline_async(**callback**, **context**)
Checks if the GPS is currently reachable, without raising any error (asynchronous version).

gps→load(**msValidity**)
Preloads the GPS cache with a specified validity duration.

gps→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

gps→load_async(**msValidity**, **callback**, **context**)
Preloads the GPS cache with a specified validity duration (asynchronous version).

gps→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

gps→nextGps()

Continues the enumeration of GPS started using `yFirstGps()`.

gps→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

gps→set_coordSystem(**newval**)

Changes the representation system used for positioning data.

gps→set_logicalName(**newval**)

Changes the logical name of the GPS.

gps→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

gps→set_utcOffset(**newval**)

Changes the number of seconds between current time and UTC time (time zone).

gps→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

gps→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGps.FindGps() YGps
yFindGps()yFindGps()

Retrieves a GPS for a given identifier.

function **FindGps**(**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 GPS is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGps.isOnline()` to test if the GPS is indeed online at a given time. In case of ambiguity when looking for a GPS 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the GPS

Returns :

a `YGps` object allowing you to drive the GPS.

YGps.FindGpsInContext() YGps
yFindGpsInContext()

Retrieves a GPS for a given identifier in a YAPI context.

function **FindGpsInContext**(**yctx**, **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 GPS is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGps.isOnline()` to test if the GPS is indeed online at a given time. In case of ambiguity when looking for a GPS 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 :

yctx a YAPI context

func a string that uniquely characterizes the GPS

Returns :

a `YGps` object allowing you to drive the GPS.

YGps.FirstGps() YGps
yFirstGps()yFirstGps()

Starts the enumeration of GPS currently accessible.

function **FirstGps**( )

Use the method `YGps.nextGps()` to iterate on next GPS.

Returns :

a pointer to a `YGps` object, corresponding to the first GPS currently online, or a `null` pointer if there are none.

YGps.FirstGpsInContext() YGps
yFirstGpsInContext()

Starts the enumeration of GPS currently accessible.

function **FirstGpsInContext**(**yctx**)

Use the method `YGps.nextGps()` to iterate on next GPS.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YGps` object, corresponding to the first GPS currently online, or a `null` pointer if there are none.

gps→clearCache()gps.clearCache() YGps

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the GPS attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

gps→describe()gps.describe() YGps

Returns a short text that describes unambiguously the instance of the GPS in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the GPS (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

gps→get_advertisedValue() YGps
gps→advertisedValue()gps.get_advertisedValue()

Returns the current value of the GPS (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the GPS (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

gps→get_altitude() YGps
gps→altitude()gps.get_altitude()

Returns the current altitude.

```
function get_altitude( )
```

Beware: GPS technology is very inaccurate regarding altitude.

Returns :
a floating point number corresponding to the current altitude

On failure, throws an exception or returns `Y_ALTITUDE_INVALID`.

gps→get_coordSystem() YGps
gps→coordSystem()gps.get_coordSystem()

Returns the representation system used for positioning data.

function **get_coordSystem**( )

Returns :

a value among `Y_COORDSYSTEM_GPS_DMS`, `Y_COORDSYSTEM_GPS_DM` and `Y_COORDSYSTEM_GPS_D` corresponding to the representation system used for positioning data

On failure, throws an exception or returns `Y_COORDSYSTEM_INVALID`.

gps→get_dateTime() YGps
gps→dateTime()gps.get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss".

function **get_dateTime**( )

Returns :
a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns `Y_DATETIME_INVALID`.

gps→get_dilution() YGps
gps→dilution()gps.get_dilution()

Returns the current horizontal dilution of precision, the smaller that number is, the better .

function **get_dilution**( )

Returns :

a floating point number corresponding to the current horizontal dilution of precision, the smaller that number is, the better

On failure, throws an exception or returns `Y_DILUTION_INVALID`.

gps→get_direction() YGps
gps→direction()gps.get_direction()

Returns the current move bearing in degrees, zero is the true (geographic) north.

function **get_direction**( )

Returns :

a floating point number corresponding to the current move bearing in degrees, zero is the true (geographic) north

On failure, throws an exception or returns `Y_DIRECTION_INVALID`.

gps→get_errorMessage() YGps
gps→errorMessage()gps.get_errorMessage()

Returns the error message of the latest error with the GPS.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the GPS object

gps→get_errorType() YGps
gps→errorType()gps.get_errorType()

Returns the numerical error code of the latest error with the GPS.

```
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 the GPS object

gps→get_friendlyName() YGps
gps→friendlyName()gps.get_friendlyName()

Returns a global identifier of the GPS in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the GPS if they are defined, otherwise the serial number of the module and the hardware identifier of the GPS (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the GPS using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

gps→get_functionDescriptor() YGps
gps→functionDescriptor()
gps.get_functionDescriptor()

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.

gps→get_functionId() YGps
gps→functionId()gps.get_functionId()

Returns the hardware identifier of the GPS, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the GPS (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

gps→get_groundSpeed() YGps
gps→groundSpeed()gps.get_groundSpeed()

Returns the current ground speed in Km/h.

function **get_groundSpeed**( )

Returns :
a floating point number corresponding to the current ground speed in Km/h

On failure, throws an exception or returns `Y_GROUNDSPEED_INVALID`.

gps→get_hardwareId() YGps
gps→hardwareId()gps.get_hardwareId()

Returns the unique hardware identifier of the GPS in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the GPS (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the GPS (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

gps→get_isFixed() YGps
gps→isFixed()gps.get_isFixed()

Returns TRUE if the receiver has found enough satellites to work.

function **get_isFixed**( )

Returns :

either `Y_ISFIXED_FALSE` or `Y_ISFIXED_TRUE`, according to TRUE if the receiver has found enough satellites to work

On failure, throws an exception or returns `Y_ISFIXED_INVALID`.

gps→get_latitude() YGps
gps→latitude()gps.get_latitude()

Returns the current latitude.

function **get_latitude**( )

Returns :
a string corresponding to the current latitude

On failure, throws an exception or returns `Y_LATITUDE_INVALID`.

gps→get_logicalName() YGps
gps→logicalName()gps.get_logicalName()

Returns the logical name of the GPS.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the GPS.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

gps→get_longitude() YGps
gps→longitude()gps.get_longitude()

Returns the current longitude.

function **get_longitude**( )

Returns :
a string corresponding to the current longitude

On failure, throws an exception or returns `Y_LONGITUDE_INVALID`.

gps→get_module() YGps
gps→module()gps.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

gps→get_satCount() YGps
gps→satCount()gps.get_satCount()

Returns the count of visible satellites.

function **get_satCount**( )

Returns :
an integer corresponding to the count of visible satellites

On failure, throws an exception or returns `Y_SATCOUNT_INVALID`.

gps→get_unixTime() YGps
gps→unixTime()gps.get_unixTime()

Returns the current time in Unix format (number of seconds elapsed since Jan 1st, 1970).

function **get_unixTime**( )

Returns :
an integer corresponding to the current time in Unix format (number of seconds elapsed since Jan 1st, 1970)

On failure, throws an exception or returns `Y_UNIXTIME_INVALID`.

gps→get_userData() YGps
gps→userData()gps.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.

gps→get_utcOffset() YGps
gps→utcOffset()gps.get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

function **get_utcOffset**( )

Returns :
an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns `Y_UTCOFFSET_INVALID`.

gps→isOnline()gps.isOnline() YGps

Checks if the GPS is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the GPS 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 GPS.

Returns :

`true` if the GPS can be reached, and `false` otherwise

gps→load()gps.load() YGps

Preloads the GPS 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.

gps→loadAttribute()gps.loadAttribute() YGps

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

gps→muteValueCallbacks() YGps
gps.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

gps→nextGps()gps.nextGps() YGps

Continues the enumeration of GPS started using `yFirstGps()`.

function **nextGps**( )

Returns :

a pointer to a `YGps` object, corresponding to a GPS currently online, or a `null` pointer if there are no more GPS to enumerate.

gps→registerValueCallback() YGps
gps.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.

gps→set_coordSystem() YGps
gps→setCoordSystem()gps.set_coordSystem()

Changes the representation system used for positioning data.

function **set_coordSystem**(**newval**)

Parameters :

newval a value among `Y_COORDSYSTEM_GPS_DMS`, `Y_COORDSYSTEM_GPS_DM` and `Y_COORDSYSTEM_GPS_D` corresponding to the representation system used for positioning data

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gps→set_logicalName() YGps
gps→setLogicalName()gps.set_logicalName()

Changes the logical name of the GPS.

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 GPS.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gps→set_userData() YGps
gps→setUserData()gps.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

gps→set_utcOffset() YGps
gps→setUtcOffset()gps.set_utcOffset()

Changes the number of seconds between current time and UTC time (time zone).

function **set_utcOffset**(**newval**)

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time is automatically be updated according to the selected time zone.

Parameters :

newval an integer corresponding to the number of seconds between current time and UTC time (time zone)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gps→unmuteValueCallbacks() YGps
gps.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

gps→wait_async()gps.wait_async() YGps

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.29. GroundSpeed function interface

The Yoctopuce class YGroundSpeed allows you to read the ground speed from Yoctopuce geolocalization sensors. It inherits from the YSensor class the core functions to read measurements, register callback functions, access the autonomous datalogger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_groundspeed.js'></script>
cpp	#include "yocto_groundspeed.h"
m	#import "yocto_groundspeed.h"
pas	uses yocto_groundspeed;
vb	yocto_groundspeed.vb
cs	yocto_groundspeed.cs
java	import com.yoctopuce.YoctoAPI.YGroundSpeed;
uwp	import com.yoctopuce.YoctoAPI.YGroundSpeed;
py	from yocto_groundspeed import *
php	require_once('yocto_groundspeed.php');
es	in HTML: <script src="../../lib/yocto_groundspeed.js"></script> in node.js: require('yoctolib-es2017/yocto_groundspeed.js');

Global functions

yFindGroundSpeed(**func**)

Retrieves a ground speed sensor for a given identifier.

yFindGroundSpeedInContext(**yctx**, **func**)

Retrieves a ground speed sensor for a given identifier in a YAPI context.

yFirstGroundSpeed()

Starts the enumeration of ground speed sensors currently accessible.

yFirstGroundSpeedInContext(**yctx**)

Starts the enumeration of ground speed sensors currently accessible.

YGroundSpeed methods

groundspeed→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

groundspeed→clearCache()

Invalidates the cache.

groundspeed→describe()

Returns a short text that describes unambiguously the instance of the ground speed sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

groundspeed→get_advertisedValue()

Returns the current value of the ground speed sensor (no more than 6 characters).

groundspeed→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in km/h, as a floating point number.

groundspeed→get_currentValue()

Returns the current value of the ground speed, in km/h, as a floating point number.

groundspeed→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

groundspeed→get_errorMessage()

Returns the error message of the latest error with the ground speed sensor.

groundspeed→get_errorType()

Returns the numerical error code of the latest error with the ground speed sensor.

groundspeed→get_friendlyName()

Returns a global identifier of the ground speed sensor in the format `MODULE_NAME.FUNCTION_NAME`.

groundspeed→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

groundspeed→get_functionId()

Returns the hardware identifier of the ground speed sensor, without reference to the module.

groundspeed→get_hardwareId()

Returns the unique hardware identifier of the ground speed sensor in the form `SERIAL.FUNCTIONID`.

groundspeed→get_highestValue()

Returns the maximal value observed for the ground speed since the device was started.

groundspeed→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

groundspeed→get_logicalName()

Returns the logical name of the ground speed sensor.

groundspeed→get_lowestValue()

Returns the minimal value observed for the ground speed since the device was started.

groundspeed→get_module()

Gets the `YModule` object for the device on which the function is located.

groundspeed→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

groundspeed→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

groundspeed→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

groundspeed→get_resolution()

Returns the resolution of the measured values.

groundspeed→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

groundspeed→get_unit()

Returns the measuring unit for the ground speed.

groundspeed→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

groundspeed→isOnline()

Checks if the ground speed sensor is currently reachable, without raising any error.

groundspeed→isOnline_async(callback, context)

Checks if the ground speed sensor is currently reachable, without raising any error (asynchronous version).

groundspeed→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

groundspeed→load(msValidity)

Preloads the ground speed sensor cache with a specified validity duration.

groundspeed→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

groundspeed→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

groundspeed→load_async(msValidity, callback, context)

Preloads the ground speed sensor cache with a specified validity duration (asynchronous version).

groundspeed→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

groundspeed→nextGroundSpeed()

Continues the enumeration of ground speed sensors started using `yFirstGroundSpeed()`.

groundspeed→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

groundspeed→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

groundspeed→set_highestValue(newval)

Changes the recorded maximal value observed.

groundspeed→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

groundspeed→set_logicalName(newval)

Changes the logical name of the ground speed sensor.

groundspeed→set_lowestValue(newval)

Changes the recorded minimal value observed.

groundspeed→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

groundspeed→set_resolution(newval)

Changes the resolution of the measured physical values.

groundspeed→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

groundspeed→startDataLogger()

Starts the data logger on the device.

groundspeed→stopDataLogger()

Stops the datalogger on the device.

groundspeed→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

groundspeed→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGroundSpeed.FindGroundSpeed() YGroundSpeed yFindGroundSpeed()yFindGroundSpeed()

Retrieves a ground speed sensor for a given identifier.

```
function FindGroundSpeed( 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 ground speed sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGroundSpeed.isOnline()` to test if the ground speed sensor is indeed online at a given time. In case of ambiguity when looking for a ground speed 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the ground speed sensor

Returns :

a `YGroundSpeed` object allowing you to drive the ground speed sensor.

YGroundSpeed.FindGroundSpeedInContext() YGroundSpeed yFindGroundSpeedInContext()

Retrieves a ground speed sensor for a given identifier in a YAPI context.

function **FindGroundSpeedInContext**(**yctx**, **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 ground speed sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGroundSpeed.isOnline()` to test if the ground speed sensor is indeed online at a given time. In case of ambiguity when looking for a ground speed 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 :

yctx a YAPI context

func a string that uniquely characterizes the ground speed sensor

Returns :

a `YGroundSpeed` object allowing you to drive the ground speed sensor.

YGroundSpeed.FirstGroundSpeed() YGroundSpeed yFirstGroundSpeed()yFirstGroundSpeed()

Starts the enumeration of ground speed sensors currently accessible.

function **FirstGroundSpeed**( )

Use the method `YGroundSpeed.nextGroundSpeed()` to iterate on next ground speed sensors.

Returns :

a pointer to a `YGroundSpeed` object, corresponding to the first ground speed sensor currently online, or a `null` pointer if there are none.

YGroundSpeed.FirstGroundSpeedInContext() YGroundSpeed yFirstGroundSpeedInContext()

Starts the enumeration of ground speed sensors currently accessible.

function **FirstGroundSpeedInContext**(**yctx**)

Use the method `YGroundSpeed.nextGroundSpeed()` to iterate on next ground speed sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YGroundSpeed` object, corresponding to the first ground speed sensor currently online, or a `null` pointer if there are none.

groundspeed→calibrateFromPoints() YGroundSpeed
groundspeed.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.

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.

groundspeed→clearCache() YGroundSpeed
groundspeed.clearCache()

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the ground speed sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

groundspeed→describe()groundspeed.describe() YGroundSpeed

Returns a short text that describes unambiguously the instance of the ground speed sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the ground speed sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

groundspeed→get_advertisedValue() YGroundSpeed
groundspeed→advertisedValue()
groundspeed.get_advertisedValue()

Returns the current value of the ground speed sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the ground speed sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

groundspeed→get_currentRawValue() YGroundSpeed
groundspeed→currentRawValue()
groundspeed.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in km/h, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in km/h, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

groundspeed→get_currentValue() YGroundSpeed
groundspeed→currentValue()
groundspeed.get_currentValue()

Returns the current value of the ground speed, in km/h, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the ground speed, in km/h, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

groundspeed→get_dataLogger() YGroundSpeed
groundspeed→dataLogger()
groundspeed.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

groundspeed→get_errorMessage() YGroundSpeed
groundspeed→errorMessage()
groundspeed.get_errorMessage()

Returns the error message of the latest error with the ground speed sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the ground speed sensor object

groundspeed→get_errorType() YGroundSpeed
groundspeed→errorType()
groundspeed.get_errorType()

Returns the numerical error code of the latest error with the ground speed sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the ground speed sensor object

groundspeed→get_friendlyName() YGroundSpeed
groundspeed→friendlyName()
groundspeed.get_friendlyName()

Returns a global identifier of the ground speed sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the ground speed sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the ground speed sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the ground speed sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

groundspeed→get_functionDescriptor() YGroundSpeed
groundspeed→functionDescriptor()
groundspeed.get_functionDescriptor()

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.

groundspeed→get_functionId() YGroundSpeed
groundspeed→functionId()
groundspeed.get_functionId()

Returns the hardware identifier of the ground speed sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the ground speed sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

groundspeed→get_hardwareId() YGroundSpeed
groundspeed→hardwareId()
groundspeed.get_hardwareId()

Returns the unique hardware identifier of the ground speed sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the ground speed sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the ground speed sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

groundspeed→get_highestValue() YGroundSpeed
groundspeed→highestValue()
groundspeed.get_highestValue()

Returns the maximal value observed for the ground speed since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the ground speed since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

groundspeed→get_logFrequency() YGroundSpeed
groundspeed→logFrequency()
groundspeed.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

groundspeed→get_logicalName() YGroundSpeed
groundspeed→logicalName()
groundspeed.get_logicalName()

Returns the logical name of the ground speed sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the ground speed sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

groundspeed→get_lowestValue() YGroundSpeed
groundspeed→lowestValue()
groundspeed.get_lowestValue()

Returns the minimal value observed for the ground speed since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the ground speed since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

groundspeed→get_module() YGroundSpeed
groundspeed→module()groundspeed.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

groundspeed→get_recordedData() YGroundSpeed
groundspeed→recordedData()
groundspeed.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

groundspeed→get_reportFrequency() YGroundSpeed
groundspeed→reportFrequency()
groundspeed.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

groundspeed→get_resolution() YGroundSpeed
groundspeed→resolution()
groundspeed.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

groundspeed→get_sensorState() YGroundSpeed
groundspeed→sensorState()
groundspeed.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :
an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

groundspeed→get_unit() YGroundSpeed
groundspeed→unit()groundspeed.get_unit()

Returns the measuring unit for the ground speed.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the ground speed

On failure, throws an exception or returns `Y_UNIT_INVALID`.

groundspeed→get_userData() YGroundSpeed
groundspeed→userData()
groundspeed.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.

groundspeed→isOnline()groundspeed.isOnline() YGroundSpeed

Checks if the ground speed sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the ground speed sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the ground speed sensor.

Returns :

`true` if the ground speed sensor can be reached, and `false` otherwise

groundspeed→load()groundspeed.load() YGroundSpeed

Preloads the ground speed sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

groundspeed→loadAttribute() YGroundSpeed
groundspeed.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

groundspeed→loadCalibrationPoints() YGroundSpeed
groundspeed.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→muteValueCallbacks() YGroundSpeed
groundspeed.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→nextGroundSpeed() YGroundSpeed
groundspeed.nextGroundSpeed()

Continues the enumeration of ground speed sensors started using `yFirstGroundSpeed()`.

function **nextGroundSpeed**( )

Returns :

a pointer to a `YGroundSpeed` object, corresponding to a ground speed sensor currently online, or a `null` pointer if there are no more ground speed sensors to enumerate.

groundspeed→registerTimedReportCallback() YGroundSpeed
groundspeed.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

groundspeed→registerValueCallback() YGroundSpeed
groundspeed.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.

groundspeed→set_highestValue() YGroundSpeed
groundspeed→setHighestValue()
groundspeed.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.

groundspeed→set_logFrequency() YGroundSpeed
groundspeed→setLogFrequency()
groundspeed.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→set_logicalName() YGroundSpeed
groundspeed→setLogicalName()
groundspeed.set_logicalName()

Changes the logical name of the ground speed 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 ground speed sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→set_lowestValue() YGroundSpeed
groundspeed→setLowestValue()
groundspeed.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.

groundspeed→set_reportFrequency() YGroundSpeed
groundspeed→setReportFrequency()
groundspeed.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→set_resolution() YGroundSpeed
groundspeed→setResolution()
groundspeed.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→set_userData() YGroundSpeed
groundspeed→setUserData()
groundspeed.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

groundspeed→startDataLogger() YGroundSpeed
groundspeed.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

groundspeed→stopDataLogger() YGroundSpeed
groundspeed.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

groundspeed→unmuteValueCallbacks() YGroundSpeed
groundspeed.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

groundspeed→wait_async() YGroundSpeed
groundspeed.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.30. Gyroscope function interface

The YSensor class is the parent class for all Yoctopuce sensors. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provide a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButton class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

```
js    <script type='text/javascript' src='yocto_gyro.js'></script>
cpp   #include "yocto_gyro.h"
m     #import "yocto_gyro.h"
pas   uses yocto_gyro;
vb    yocto_gyro.vb
cs    yocto_gyro.cs
java  import com.yoctopuce.YoctoAPI.YGyro;
uwp   import com.yoctopuce.YoctoAPI.YGyro;
py    from yocto_gyro import *
php   require_once('yocto_gyro.php');
es    in HTML: <script src="../../lib/yocto_gyro.js"></script>
      in node.js: require('yoctolib-es2017/yocto_gyro.js');
```

Global functions
yFindGyro(**func**) Retrieves a gyroscope for a given identifier.
yFindGyroInContext(**yctx**, **func**) Retrieves a gyroscope for a given identifier in a YAPI context.
yFirstGyro() Starts the enumeration of gyroscopes currently accessible.
yFirstGyroInContext(**yctx**) Starts the enumeration of gyroscopes currently accessible.
YGyro methods
gyro→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
gyro→clearCache() Invalidates the cache.
gyro→describe() Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
gyro→get_advertisedValue() Returns the current value of the gyroscope (no more than 6 characters).
gyro→get_bandwidth() Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).
gyro→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.
gyro→get_currentValue()

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

gyro→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

gyro→get_errorMessage()

Returns the error message of the latest error with the gyroscope.

gyro→get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

gyro→get_friendlyName()

Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.

gyro→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

gyro→get_functionId()

Returns the hardware identifier of the gyroscope, without reference to the module.

gyro→get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form `SERIAL.FUNCTIONID`.

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

gyro→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

gyro→get_logicalName()

Returns the logical name of the gyroscope.

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

gyro→get_module()

Gets the `YModule` object for the device on which the function is located.

gyro→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionW()

Returns the `w` component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionX()

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionY()

Returns the `y` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionZ()

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

gyro→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

gyro→get_resolution()

Returns the resolution of the measured values.

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

gyro→get_unit()

Returns the measuring unit for the angular velocity.

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

gyro→get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

gyro→get_yValue()

Returns the angular velocity around the Y axis of the device, as a floating point number.

gyro→get_zValue()

Returns the angular velocity around the Z axis of the device, as a floating point number.

gyro→isOnline()

Checks if the gyroscope is currently reachable, without raising any error.

gyro→isOnline_async(**callback**, **context**)

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

gyro→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

gyro→load(**msValidity**)

Preloads the gyroscope cache with a specified validity duration.

gyro→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

gyro→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

gyro→load_async(**msValidity**, **callback**, **context**)

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

gyro→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

gyro→nextGyro()

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

gyro→registerAnglesCallback(**callback**)

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerQuaternionCallback(**callback**)

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

gyro→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

gyro→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

gyro→set_highestValue(newval)

Changes the recorded maximal value observed.

gyro→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

gyro→set_logicalName(newval)

Changes the logical name of the gyroscope.

gyro→set_lowestValue(newval)

Changes the recorded minimal value observed.

gyro→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

gyro→set_resolution(newval)

Changes the resolution of the measured physical values.

gyro→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

gyro→startDataLogger()

Starts the data logger on the device.

gyro→stopDataLogger()

Stops the datalogger on the device.

gyro→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

gyro→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGyro.FindGyro() YGyro
yFindGyro()yFindGyro()

Retrieves a gyroscope for a given identifier.

function **FindGyro**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the gyroscope is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGyro.isOnline()` to test if the gyroscope is indeed online at a given time. In case of ambiguity when looking for a gyroscope by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the gyroscope

Returns :

a `YGyro` object allowing you to drive the gyroscope.

YGyro.FindGyroInContext() YGyro
yFindGyroInContext()

Retrieves a gyroscope for a given identifier in a YAPI context.

function **FindGyroInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the gyroscope is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGyro.isOnline()` to test if the gyroscope is indeed online at a given time. In case of ambiguity when looking for a gyroscope by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the gyroscope

Returns :

a `YGyro` object allowing you to drive the gyroscope.

YGyro.FirstGyro() YGyro yFirstGyro()yFirstGyro()

Starts the enumeration of gyroscopes currently accessible.

function **FirstGyro**( )

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

Returns :

a pointer to a `YGyro` object, corresponding to the first gyro currently online, or a `null` pointer if there are none.

YGyro.FirstGyroInContext() YGyro yFirstGyroInContext()

Starts the enumeration of gyroscopes currently accessible.

function **FirstGyroInContext**(**yctx**)

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YGyro` object, corresponding to the first gyro currently online, or a `null` pointer if there are none.

gyro→calibrateFromPoints() YGyro
gyro.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→clearCache()gyro.clearCache() YGyro

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the gyroscope attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

gyro→describe()gyro.describe() YGyro

Returns a short text that describes unambiguously the instance of the gyroscope in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the gyroscope (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

gyro→get_advertisedValue() YGyro
gyro→advertisedValue()gyro.get_advertisedValue()

Returns the current value of the gyroscope (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the gyroscope (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

gyro→get_bandwidth() YGyro
gyro→bandwidth()gyro.get_bandwidth()

Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **get_bandwidth**( )

Returns :
an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

On failure, throws an exception or returns Y_BANDWIDTH_INVALID.

gyro→get_currentRawValue() YGyro
gyro→currentRawValue()gyro.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

gyro→get_currentValue() YGyro
gyro→currentValue()gyro.get_currentValue()

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the angular velocity, in degrees per second, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

gyro→get_dataLogger() YGyro
gyro→dataLogger()gyro.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

gyro→get_errorMessage() YGyro
gyro→errorMessage()gyro.get_errorMessage()

Returns the error message of the latest error with the gyroscope.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the gyroscope object

gyro→get_errorType() YGyro
gyro→errorType()gyro.get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the gyroscope object

gyro→get_friendlyName() YGyro
gyro→friendlyName()gyro.get_friendlyName()

Returns a global identifier of the gyroscope in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the gyroscope if they are defined, otherwise the serial number of the module and the hardware identifier of the gyroscope (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the gyroscope using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

gyro→get_functionDescriptor() YGyro
gyro→functionDescriptor()
gyro.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

gyro→get_functionId() YGyro
gyro→functionId()gyro.get_functionId()

Returns the hardware identifier of the gyroscope, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the gyroscope (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

gyro→get_hardwareId() YGyro
gyro→hardwareId()gyro.get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the gyroscope (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the gyroscope (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

gyro→get_heading() YGyro
gyro→heading()gyro.get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_heading**( )

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :
a floating-point number corresponding to heading in degrees, between 0 and 360.

gyro→get_highestValue() YGyro
gyro→highestValue()gyro.get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

gyro→get_logFrequency() YGyro
gyro→logFrequency()gyro.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

gyro→get_logicalName() YGyro
gyro→logicalName()gyro.get_logicalName()

Returns the logical name of the gyroscope.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the gyroscope.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

gyro→get_lowestValue() YGyro
gyro→lowestValue()gyro.get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

gyro→get_module() YGyro
gyro→module()gyro.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

gyro→get_pitch() YGyro
gyro→pitch()gyro.get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_pitch**( )

The axis corresponding to the pitch angle can be mapped to any of the device X, Y or Z physical directions using methods of the class YRefFrame.

Returns :
a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

gyro→get_quaternionW() YGyro
gyro→quaternionW()gyro.get_quaternionW()

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_quaternionW**( )

Returns :

a floating-point number corresponding to the w component of the quaternion.

gyro→get_quaternionX() YGyro
gyro→quaternionX()gyro.get_quaternionX()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_quaternionX**( )

The x component is mostly correlated with rotations on the roll axis.

Returns :

a floating-point number corresponding to the x component of the quaternion.

gyro→get_quaternionY() YGyro
gyro→quaternionY()gyro.get_quaternionY()

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_quaternionY**( )

The y component is mostly correlated with rotations on the pitch axis.

Returns :

a floating-point number corresponding to the y component of the quaternion.

gyro→get_quaternionZ() YGyro
gyro→quaternionZ()gyro.get_quaternionZ()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_quaternionZ**( )

The x component is mostly correlated with changes of heading.

Returns :

a floating-point number corresponding to the z component of the quaternion.

gyro→get_recordedData() YGyro
gyro→recordedData()gyro.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

gyro→get_reportFrequency() YGyro
gyro→reportFrequency()gyro.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

gyro→get_resolution() YGyro
gyro→resolution()gyro.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

gyro→get_roll() YGyro
gyro→roll()gyro.get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

function **get_roll**( )

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :
a floating-point number corresponding to roll angle in degrees, between -180 and +180.

gyro→get_sensorState() YGyro
gyro→sensorState()gyro.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

gyro→get_unit() YGyro
gyro→unit()gyro.get_unit()

Returns the measuring unit for the angular velocity.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

gyro→get_userData() YGyro
gyro→userData()gyro.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.

gyro→get_xValue() YGyro
gyro→xValue()gyro.get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

function **get_xValue**( )

Returns :

a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

gyro→get_yValue() YGyro
gyro→yValue()gyro.get_yValue()

Returns the angular velocity around the Y axis of the device, as a floating point number.

function **get_yValue**( )

Returns :

a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

gyro→get_zValue() YGyro
gyro→zValue()gyro.get_zValue()

Returns the angular velocity around the Z axis of the device, as a floating point number.

function **get_zValue**( )

Returns :

a floating point number corresponding to the angular velocity around the Z axis of the device, as a floating point number

On failure, throws an exception or returns `Y_ZVALUE_INVALID`.

gyro→isOnline()gyro.isOnline() YGyro

Checks if the gyroscope is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

Returns :

`true` if the gyroscope can be reached, and `false` otherwise

gyro→load()gyro.load() YGyro

Preloads the gyroscope cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→loadAttribute()gyro.loadAttribute() YGyro

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

gyro→loadCalibrationPoints() YGyro
gyro.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→muteValueCallbacks() YGyro
gyro.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→nextGyro()gyro.nextGyro() YGyro

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

function **nextGyro**( )

Returns :

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.

gyro→registerAnglesCallback() YGyro
gyro.registerAnglesCallback()

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

function **registerAnglesCallback**(**callback**)

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGyro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

gyro→registerQuaternionCallback() YGyro
gyro.registerQuaternionCallback()

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

function **registerQuaternionCallback**(**callback**)

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

gyro→registerTimedReportCallback() YGyro
gyro.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

gyro→registerValueCallback() YGyro
gyro.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.

gyro→set_bandwidth() YGyro
gyro→setBandwidth()gyro.set_bandwidth()

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **set_bandwidth**(**newval**)

When the frequency is lower, the device performs averaging.

Parameters :

newval an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_highestValue() YGyro
gyro→setHighestValue()gyro.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.

gyro→set_logFrequency() YGyro
gyro→setLogFrequency()gyro.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logicalName() YGyro
gyro→setLogicalName()gyro.set_logicalName()

Changes the logical name of the gyroscope.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the gyroscope.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_lowestValue() YGyro
gyro→setLowestValue()gyro.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.

gyro→set_reportFrequency() YGyro
gyro→setReportFrequency()
gyro.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_resolution() YGyro
gyro→setResolution()gyro.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_userData() YGyro
gyro→setUserData()gyro.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

gyro→startDataLogger()gyro.startDataLogger() YGyro

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

gyro→stopDataLogger()gyro.stopDataLogger() YGyro

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

gyro→unmuteValueCallbacks() YGyro
gyro.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→wait_async()gyro.wait_async() YGyro

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.31. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_hubport.js'></script>
cpp	#include "yocto_hubport.h"
m	#import "yocto_hubport.h"
pas	uses yocto_hubport;
vb	yocto_hubport.vb
cs	yocto_hubport.cs
java	import com.yoctopuce.YoctoAPI.YHubPort;
uwp	import com.yoctopuce.YoctoAPI.YHubPort;
py	from yocto_hubport import *
php	require_once('yocto_hubport.php');
es	in HTML: <script src="../../lib/yocto_hubport.js"></script> in node.js: require('yoctolib-es2017/yocto_hubport.js');

Global functions	
yFindHubPort(**func**)	Retrieves a Yocto-hub port for a given identifier.
yFindHubPortInContext(**yctx**, **func**)	Retrieves a Yocto-hub port for a given identifier in a YAPI context.
yFirstHubPort()	Starts the enumeration of Yocto-hub ports currently accessible.
yFirstHubPortInContext(**yctx**)	Starts the enumeration of Yocto-hub ports currently accessible.

`YHubPort` methods	
hubport→clearCache()	Invalidates the cache.
hubport→describe()	Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
hubport→get_advertisedValue()	Returns the current value of the Yocto-hub port (no more than 6 characters).
hubport→get_baudRate()	Returns the current baud rate used by this Yocto-hub port, in kbps.
hubport→get_enabled()	Returns true if the Yocto-hub port is powered, false otherwise.
hubport→get_errorMessage()	Returns the error message of the latest error with the Yocto-hub port.
hubport→get_errorType()	Returns the numerical error code of the latest error with the Yocto-hub port.
hubport→get_friendlyName()	Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.
hubport→get_functionDescriptor()	

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

hubport→get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

hubport→get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form SERIAL.FUNCTIONID.

hubport→get_logicalName()

Returns the logical name of the Yocto-hub port.

hubport→get_module()

Gets the YModule object for the device on which the function is located.

hubport→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

hubport→get_portState()

Returns the current state of the Yocto-hub port.

hubport→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

hubport→isOnline()

Checks if the Yocto-hub port is currently reachable, without raising any error.

hubport→isOnline_async(callback, context)

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

hubport→load(msValidity)

Preloads the Yocto-hub port cache with a specified validity duration.

hubport→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

hubport→load_async(msValidity, callback, context)

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

hubport→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

hubport→nextHubPort()

Continues the enumeration of Yocto-hub ports started using yFirstHubPort().

hubport→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

hubport→set_enabled(newval)

Changes the activation of the Yocto-hub port.

hubport→set_logicalName(newval)

Changes the logical name of the Yocto-hub port.

hubport→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

hubport→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

hubport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHubPort.FindHubPort() YHubPort yFindHubPort()yFindHubPort()

Retrieves a Yocto-hub port for a given identifier.

function **FindHubPort**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Yocto-hub port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHubPort.isOnline()` to test if the Yocto-hub port is indeed online at a given time. In case of ambiguity when looking for a Yocto-hub port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the Yocto-hub port

Returns :

a `YHubPort` object allowing you to drive the Yocto-hub port.

YHubPort.FindHubPortInContext() YHubPort yFindHubPortInContext()

Retrieves a Yocto-hub port for a given identifier in a YAPI context.

function **FindHubPortInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the Yocto-hub port

Returns :

a `YHubPort` object allowing you to drive the Yocto-hub port.

YHubPort.FirstHubPort() YHubPort yFirstHubPort()yFirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

function **FirstHubPort**( )

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

Returns :

a pointer to a `YHubPort` object, corresponding to the first Yocto-hub port currently online, or a `null` pointer if there are none.

YHubPort.FirstHubPortInContext() yFirstHubPortInContext()

YHubPort

Starts the enumeration of Yocto-hub ports currently accessible.

function **FirstHubPortInContext**(**yctx**)

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

Parameters :

yctx a YAPI context.

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→clearCache()hubport.clearCache() YHubPort

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the Yocto-hub port attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

hubport→describe()hubport.describe() YHubPort

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the Yocto-hub port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

hubport→get_advertisedValue() YHubPort
hubport→advertisedValue()
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() YHubPort
hubport→baudRate()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 encountered.

Returns :
an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns `Y_BAUDRATE_INVALID`.

hubport→get_enabled() YHubPort
hubport→enabled()hubport.get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

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() YHubPort
hubport→errorMessage()hubport.get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the Yocto-hub port object

hubport→get_errorType() YHubPort
hubport→errorType()hubport.get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the Yocto-hub port object

hubport→get_friendlyName() YHubPort
hubport→friendlyName()hubport.get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the Yocto-hub port if they are defined, otherwise the serial number of the module and the hardware identifier of the Yocto-hub port (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the Yocto-hub port using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

hubport→get_functionDescriptor() YHubPort
hubport→functionDescriptor()
hubport.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

hubport→get_functionId() YHubPort
hubport→functionId()hubport.get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the Yocto-hub port (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

hubport→get_hardwareId() YHubPort
hubport→hardwareId()hubport.get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Yocto-hub port (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the Yocto-hub port (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

hubport→get_logicalName() YHubPort
hubport→logicalName()hubport.get_logicalName()

Returns the logical name of the Yocto-hub port.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the Yocto-hub port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

hubport→get_module() YHubPort
hubport→module()hubport.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

hubport→get_portState() YHubPort
hubport→portState()hubport.get_portState()

Returns the current state of the Yocto-hub port.

function **get_portState**( )

Returns :

a value among Y_PORTSTATE_OFF, Y_PORTSTATE_OVRLD, Y_PORTSTATE_ON, Y_PORTSTATE_RUN and Y_PORTSTATE_PROG corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns Y_PORTSTATE_INVALID.

hubport→get_userData() YHubPort
hubport→userData()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()hubport.isOnline() YHubPort

Checks if the Yocto-hub port is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Yocto-hub port.

Returns :
`true` if the Yocto-hub port can be reached, and `false` otherwise

hubport→load()hubport.load() YHubPort

Preloads the Yocto-hub port cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()hubport.loadAttribute() YHubPort

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

hubport→muteValueCallbacks() YHubPort
hubport.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→nextHubPort()hubport.nextHubPort() YHubPort

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

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() YHubPort
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() YHubPort
hubport→setEnabled()hubport.set_enabled()

Changes the activation of the Yocto-hub port.

function **set_enabled**(**newval**)

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the activation of the Yocto-hub port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→set_logicalName() YHubPort
hubport→setLogicalName()
hubport.set_logicalName()

Changes the logical name of the Yocto-hub port.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Yocto-hub port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→set_userData() YHubPort
hubport→setUserData()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

hubport→unmuteValueCallbacks() YHubPort
hubport.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→wait_async()hubport.wait_async() YHubPort

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.32. Humidity function interface

The Yoctopuce class YHumidity allows you to read and configure Yoctopuce humidity sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_humidity.js'></script>
cpp	#include "yocto_humidity.h"
m	#import "yocto_humidity.h"
pas	uses yocto_humidity;
vb	yocto_humidity.vb
cs	yocto_humidity.cs
java	import com.yoctopuce.YoctoAPI.YHumidity;
uwp	import com.yoctopuce.YoctoAPI.YHumidity;
py	from yocto_humidity import *
php	require_once('yocto_humidity.php');
es	in HTML: <script src="../../lib/yocto_humidity.js"></script> in node.js: require('yoctolib-es2017/yocto_humidity.js');

Global functions
yFindHumidity(**func**) Retrieves a humidity sensor for a given identifier.
yFindHumidityInContext(**yctx**, **func**) Retrieves a humidity sensor for a given identifier in a YAPI context.
yFirstHumidity() Starts the enumeration of humidity sensors currently accessible.
yFirstHumidityInContext(**yctx**) 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→clearCache() Invalidates the cache.
humidity→describe() Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
humidity→get_absHum() Returns the current absolute humidity, in grams per cubic meter of air.
humidity→get_advertisedValue() Returns the current value of the humidity sensor (no more than 6 characters).
humidity→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.
humidity→get_currentValue() Returns the current value of the humidity, in %RH, as a floating point number.
humidity→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
humidity→get_errorMessage()

Returns the error message of the latest error with the humidity sensor.

humidity→get_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

humidity→get_friendlyName()

Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

humidity→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

humidity→get_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

humidity→get_hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form `SERIAL.FUNCTIONID`.

humidity→get_highestValue()

Returns the maximal value observed for the humidity since the device was started.

humidity→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

Returns the minimal value observed for the humidity since the device was started.

humidity→get_module()

Gets the `YModule` object for the device on which the function is located.

humidity→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

humidity→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

humidity→get_relHum()

Returns the current relative humidity, in per cents.

humidity→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

humidity→get_resolution()

Returns the resolution of the measured values.

humidity→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

humidity→get_unit()

Returns the measuring unit for the humidity.

humidity→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

humidity→isOnline()

Checks if the humidity sensor is currently reachable, without raising any error.

humidity→isOnline_async(**callback**, **context**)

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

humidity→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

humidity→load(msValidity)

Preloads the humidity sensor cache with a specified validity duration.

humidity→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

humidity→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

humidity→load_async(msValidity, callback, context)

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

humidity→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

humidity→nextHumidity()

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

humidity→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

humidity→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

humidity→set_highestValue(newval)

Changes the recorded maximal value observed.

humidity→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

humidity→set_logicalName(newval)

Changes the logical name of the humidity sensor.

humidity→set_lowestValue(newval)

Changes the recorded minimal value observed.

humidity→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

humidity→set_resolution(newval)

Changes the resolution of the measured physical values.

humidity→set_unit(newval)

Changes the primary unit for measuring humidity.

humidity→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

humidity→startDataLogger()

Starts the data logger on the device.

humidity→stopDataLogger()

Stops the datalogger on the device.

humidity→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

humidity→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHumidity.FindHumidity() YHumidity yFindHumidity()yFindHumidity()

Retrieves a humidity sensor for a given identifier.

```
function FindHumidity( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the humidity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHumidity.isOnline()` to test if the humidity sensor is indeed online at a given time. In case of ambiguity when looking for a humidity sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the humidity sensor

Returns :

a `YHumidity` object allowing you to drive the humidity sensor.

YHumidity.FindHumidityInContext() YHumidity yFindHumidityInContext()

Retrieves a humidity sensor for a given identifier in a YAPI context.

function **FindHumidityInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the humidity sensor

Returns :

a `YHumidity` object allowing you to drive the humidity sensor.

YHumidity.FirstHumidity() YHumidity yFirstHumidity()yFirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

function **FirstHumidity**( )

Use the method `YHumidity.nextHumidity()` to iterate on next humidity sensors.

Returns :

a pointer to a `YHumidity` object, corresponding to the first humidity sensor currently online, or a `null` pointer if there are none.

YHumidity.FirstHumidityInContext() YHumidity yFirstHumidityInContext()

Starts the enumeration of humidity sensors currently accessible.

function **FirstHumidityInContext**(**yctx**)

Use the method `YHumidity.nextHumidity()` to iterate on next humidity sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YHumidity` object, corresponding to the first humidity sensor currently online, or a `null` pointer if there are none.

humidity→calibrateFromPoints() YHumidity
humidity.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

function **calibrateFromPoints**(**rawValues**, **refValues**)

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→clearCache()humidity.clearCache() YHumidity

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the humidity sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

humidity→describe()humidity.describe() YHumidity

Returns a short text that describes unambiguously the instance of the humidity sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the humidity sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

humidity→get_absHum() YHumidity
humidity→absHum()humidity.get_absHum()

Returns the current absolute humidity, in grams per cubic meter of air.

function **get_absHum**( )

Returns :
a floating point number corresponding to the current absolute humidity, in grams per cubic meter of air

On failure, throws an exception or returns `Y_ABSHUM_INVALID`.

humidity→get_advertisedValue() YHumidity
humidity→advertisedValue()
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() YHumidity
humidity→currentRawValue()
humidity.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

humidity→get_currentValue() YHumidity
humidity→currentValue()humidity.get_currentValue()

Returns the current value of the humidity, in %RH, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the humidity, in %RH, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

humidity→get_dataLogger() YHumidity
humidity→dataLogger()humidity.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

humidity→get_errorMessage() YHumidity
humidity→errorMessage()
humidity.get_errorMessage()

Returns the error message of the latest error with the humidity sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the humidity sensor object

humidity→get_errorType() YHumidity
humidity→errorType()humidity.get_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the humidity sensor object

humidity→get_friendlyName() YHumidity
humidity→friendlyName()
humidity.get_friendlyName()

Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the humidity sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the humidity sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the humidity sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

humidity→get_functionDescriptor() YHumidity
humidity→functionDescriptor()
humidity.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

humidity→get_functionId() YHumidity
humidity→functionId()humidity.get_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the humidity sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

humidity→get_hardwareId() YHumidity
humidity→hardwareId()humidity.get_hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the humidity sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the humidity sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

humidity→get_highestValue() YHumidity
humidity→highestValue()
humidity.get_highestValue()

Returns the maximal value observed for the humidity since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the humidity since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

humidity→get_logFrequency() YHumidity
humidity→logFrequency()
humidity.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

humidity→get_logicalName() YHumidity
humidity→logicalName()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() YHumidity
humidity→lowestValue()humidity.get_lowestValue()

Returns the minimal value observed for the humidity since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the humidity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

humidity→get_module() YHumidity
humidity→module()humidity.get_module()

Gets the YModule object for the device on which the function is located.

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_recordedData() YHumidity
humidity→recordedData()
humidity.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

humidity→get_relHum() YHumidity
humidity→relHum()humidity.get_relHum()

Returns the current relative humidity, in per cents.

function **get_relHum**( )

Returns :
a floating point number corresponding to the current relative humidity, in per cents

On failure, throws an exception or returns `Y_RELHUM_INVALID`.

humidity→get_reportFrequency() YHumidity
humidity→reportFrequency()
humidity.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

humidity→get_resolution() YHumidity
humidity→resolution()humidity.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

humidity→get_sensorState() YHumidity
humidity→sensorState()humidity.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

humidity→get_unit() YHumidity
humidity→unit()humidity.get_unit()

Returns the measuring unit for the humidity.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

humidity→get_userData() YHumidity
humidity→userData()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()humidity.isOnline() YHumidity

Checks if the humidity sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the humidity sensor.

Returns :

`true` if the humidity sensor can be reached, and `false` otherwise

humidity→load()humidity.load() YHumidity

Preloads the humidity sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()humidity.loadAttribute() YHumidity

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

humidity→loadCalibrationPoints() YHumidity
humidity.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→muteValueCallbacks() YHumidity
humidity.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→nextHumidity()humidity.nextHumidity() YHumidity

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

function **nextHumidity**( )

Returns :

a pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a `null` pointer if there are no more humidity sensors to enumerate.

humidity→registerTimedReportCallback() YHumidity
humidity.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

humidity→registerValueCallback() YHumidity
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() YHumidity
humidity→setHighestValue()
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_logFrequency() YHumidity
humidity→setLogFrequency()
humidity.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logicalName() YHumidity
humidity→setLogicalName()
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() YHumidity
humidity→setLowestValue()
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_reportFrequency() YHumidity
humidity→setReportFrequency()
humidity.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_resolution() YHumidity
humidity→setResolution()humidity.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_unit() YHumidity
humidity→setUnit()humidity.set_unit()

Changes the primary unit for measuring humidity.

function **set_unit**(**newval**)

That unit is a string. If that strings starts with the letter 'g', the primary measured value is the absolute humidity, in g/m3. Otherwise, the primary measured value will be the relative humidity (RH), in per cents.

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the primary unit for measuring humidity

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_userData() YHumidity
humidity→setUserData()humidity.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

function **set_userData**(**data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

humidity→startDataLogger() YHumidity
humidity.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

humidity→stopDataLogger() YHumidity
humidity.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

humidity→unmuteValueCallbacks() YHumidity
humidity.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→wait_async()humidity.wait_async() YHumidity

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.33. Latitude function interface

The Yoctopuce class YLatitude allows you to read the latitude from Yoctopuce geolocalization sensors. It inherits from the YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_latitude.js'></script>
cpp     #include "yocto_latitude.h"
m       #import "yocto_latitude.h"
pas     uses yocto_latitude;
vb      yocto_latitude.vb
cs      yocto_latitude.cs
java    import com.yoctopuce.YoctoAPI.YLatitude;
uwp     import com.yoctopuce.YoctoAPI.YLatitude;
py      from yocto_latitude import *
php     require_once('yocto_latitude.php');
es      in HTML: <script src="../../lib/yocto_latitude.js"></script>
        in node.js: require('yoctolib-es2017/yocto_latitude.js');
```

Global functions
yFindLatitude(**func**) Retrieves a latitude sensor for a given identifier.
yFindLatitudeInContext(**yctx**, **func**) Retrieves a latitude sensor for a given identifier in a YAPI context.
yFirstLatitude() Starts the enumeration of latitude sensors currently accessible.
yFirstLatitudeInContext(**yctx**) Starts the enumeration of latitude sensors currently accessible.
YLatitude methods
latitude→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
latitude→clearCache() Invalidates the cache.
latitude→describe() Returns a short text that describes unambiguously the instance of the latitude sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
latitude→get_advertisedValue() Returns the current value of the latitude sensor (no more than 6 characters).
latitude→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.
latitude→get_currentValue() Returns the current value of the latitude, in deg/1000, as a floating point number.
latitude→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
latitude→get_errorMessage() Returns the error message of the latest error with the latitude sensor.

latitude→get_errorType()

Returns the numerical error code of the latest error with the latitude sensor.

latitude→get_friendlyName()

Returns a global identifier of the latitude sensor in the format `MODULE_NAME.FUNCTION_NAME`.

latitude→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

latitude→get_functionId()

Returns the hardware identifier of the latitude sensor, without reference to the module.

latitude→get_hardwareId()

Returns the unique hardware identifier of the latitude sensor in the form `SERIAL.FUNCTIONID`.

latitude→get_highestValue()

Returns the maximal value observed for the latitude since the device was started.

latitude→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

latitude→get_logicalName()

Returns the logical name of the latitude sensor.

latitude→get_lowestValue()

Returns the minimal value observed for the latitude since the device was started.

latitude→get_module()

Gets the `YModule` object for the device on which the function is located.

latitude→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

latitude→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

latitude→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

latitude→get_resolution()

Returns the resolution of the measured values.

latitude→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

latitude→get_unit()

Returns the measuring unit for the latitude.

latitude→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

latitude→isOnline()

Checks if the latitude sensor is currently reachable, without raising any error.

latitude→isOnline_async(callback, context)

Checks if the latitude sensor is currently reachable, without raising any error (asynchronous version).

latitude→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

latitude→load(msValidity)

Preloads the latitude sensor cache with a specified validity duration.

latitude→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

latitude→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

latitude→load_async(msValidity, callback, context)

Preloads the latitude sensor cache with a specified validity duration (asynchronous version).

latitude→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

latitude→nextLatitude()

Continues the enumeration of latitude sensors started using `yFirstLatitude()`.

latitude→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

latitude→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

latitude→set_highestValue(newval)

Changes the recorded maximal value observed.

latitude→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

latitude→set_logicalName(newval)

Changes the logical name of the latitude sensor.

latitude→set_lowestValue(newval)

Changes the recorded minimal value observed.

latitude→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

latitude→set_resolution(newval)

Changes the resolution of the measured physical values.

latitude→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

latitude→startDataLogger()

Starts the data logger on the device.

latitude→stopDataLogger()

Stops the datalogger on the device.

latitude→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

latitude→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLatitude.FindLatitude() YLatitude yFindLatitude()yFindLatitude()

Retrieves a latitude sensor for a given identifier.

```
function FindLatitude( 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 latitude sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLatitude.isOnline()` to test if the latitude sensor is indeed online at a given time. In case of ambiguity when looking for a latitude 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the latitude sensor

Returns :

a `YLatitude` object allowing you to drive the latitude sensor.

YLatitude.FindLatitudeInContext() YLatitude yFindLatitudeInContext()

Retrieves a latitude sensor for a given identifier in a YAPI context.

function **FindLatitudeInContext**(**yctx**, **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 latitude sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLatitude.isOnline()` to test if the latitude sensor is indeed online at a given time. In case of ambiguity when looking for a latitude 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 :

yctx a YAPI context

func a string that uniquely characterizes the latitude sensor

Returns :

a `YLatitude` object allowing you to drive the latitude sensor.

YLatitude.FirstLatitude() YLatitude yFirstLatitude()yFirstLatitude()

Starts the enumeration of latitude sensors currently accessible.

function **FirstLatitude**( )

Use the method `YLatitude.nextLatitude()` to iterate on next latitude sensors.

Returns :

a pointer to a `YLatitude` object, corresponding to the first latitude sensor currently online, or a `null` pointer if there are none.

YLatitude.FirstLatitudeInContext() YLatitude yFirstLatitudeInContext()

Starts the enumeration of latitude sensors currently accessible.

function **FirstLatitudeInContext**(**yctx**)

Use the method `YLatitude.nextLatitude()` to iterate on next latitude sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YLatitude` object, corresponding to the first latitude sensor currently online, or a `null` pointer if there are none.

latitude→calibrateFromPoints() YLatitude
latitude.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.

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.

latitude→clearCache()latitude.clearCache() YLatitude

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the latitude sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

latitude→describe()latitude.describe() YLatitude

Returns a short text that describes unambiguously the instance of the latitude sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the latitude sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

latitude→get_advertisedValue() YLatitude
latitude→advertisedValue()
latitude.get_advertisedValue()

Returns the current value of the latitude sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the latitude sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

latitude→get_currentRawValue() YLatitude
latitude→currentRawValue()
latitude.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

latitude→get_currentValue() YLatitude
latitude→currentValue()latitude.get_currentValue()

Returns the current value of the latitude, in deg/1000, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the latitude, in deg/1000, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

latitude→get_dataLogger() YLatitude
latitude→dataLogger()latitude.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

latitude→get_errorMessage() YLatitude
latitude→errorMessage()latitude.get_errorMessage()

Returns the error message of the latest error with the latitude sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the latitude sensor object

latitude→get_errorType() YLatitude
latitude→errorType()latitude.get_errorType()

Returns the numerical error code of the latest error with the latitude sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the latitude sensor object

latitude→get_friendlyName() YLatitude
latitude→friendlyName()latitude.get_friendlyName()

Returns a global identifier of the latitude sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the latitude sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the latitude sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the latitude sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

latitude→get_functionDescriptor() YLatitude
latitude→functionDescriptor()
latitude.get_functionDescriptor()

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.

latitude→get_functionId() YLatitude
latitude→functionId()latitude.get_functionId()

Returns the hardware identifier of the latitude sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the latitude sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

latitude→get_hardwareId() YLatitude
latitude→hardwareId()latitude.get_hardwareId()

Returns the unique hardware identifier of the latitude sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the latitude sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the latitude sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

latitude→get_highestValue() YLatitude
latitude→highestValue()latitude.get_highestValue()

Returns the maximal value observed for the latitude since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the latitude since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

latitude→get_logFrequency() YLatitude
latitude→logFrequency()latitude.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

latitude→get_logicalName() YLatitude
latitude→logicalName()latitude.get_logicalName()

Returns the logical name of the latitude sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the latitude sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

latitude→get_lowestValue() YLatitude
latitude→lowestValue()latitude.get_lowestValue()

Returns the minimal value observed for the latitude since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the latitude since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

latitude→get_module() YLatitude
latitude→module()latitude.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

latitude→get_recordedData() YLatitude
latitude→recordedData()latitude.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

latitude→get_reportFrequency() YLatitude
latitude→reportFrequency()
latitude.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

latitude→get_resolution() YLatitude
latitude→resolution()latitude.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

latitude→get_sensorState() YLatitude
latitude→sensorState()latitude.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

latitude→get_unit() YLatitude
latitude→unit()latitude.get_unit()

Returns the measuring unit for the latitude.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the latitude

On failure, throws an exception or returns `Y_UNIT_INVALID`.

latitude→get_userData() YLatitude
latitude→userData()latitude.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.

latitude→isOnline()latitude.isOnline() YLatitude

Checks if the latitude sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the latitude sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the latitude sensor.

Returns :

`true` if the latitude sensor can be reached, and `false` otherwise

latitude→load()latitude.load() YLatitude

Preloads the latitude sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

latitude→loadAttribute()latitude.loadAttribute() YLatitude

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

latitude→loadCalibrationPoints() YLatitude
latitude.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→muteValueCallbacks() YLatitude
latitude.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→nextLatitude()latitude.nextLatitude() YLatitude

Continues the enumeration of latitude sensors started using `yFirstLatitude()`.

function **nextLatitude**( )

Returns :

a pointer to a `YLatitude` object, corresponding to a latitude sensor currently online, or a `null` pointer if there are no more latitude sensors to enumerate.

latitude→registerTimedReportCallback() YLatitude
latitude.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

latitude→registerValueCallback() YLatitude
latitude.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.

latitude→set_highestValue() YLatitude
latitude→setHighestValue()
latitude.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.

latitude→set_logFrequency() YLatitude
latitude→setLogFrequency()
latitude.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→set_logicalName() YLatitude
latitude→setLogicalName()latitude.set_logicalName()

Changes the logical name of the latitude 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 latitude sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→set_lowestValue() YLatitude
latitude→setLowestValue()latitude.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.

latitude→set_reportFrequency() YLatitude
latitude→setReportFrequency()
latitude.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→set_resolution() YLatitude
latitude→setResolution()latitude.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→set_userData() YLatitude
latitude→setUserData()latitude.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

latitude→startDataLogger()latitude.startDataLogger() YLatitude

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

latitude→stopDataLogger()latitude.stopDataLogger() YLatitude

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

latitude→unmuteValueCallbacks() YLatitude
latitude.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

latitude→wait_async()latitude.wait_async() YLatitude

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.34. Led function interface

The Yoctopuce application programming interface allows you not only to drive the intensity of the LED, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_led.js'></script>
cpp	#include "yocto_led.h"
m	#import "yocto_led.h"
pas	uses yocto_led;
vb	yocto_led.vb
cs	yocto_led.cs
java	import com.yoctopuce.YoctoAPI.YLed;
uwp	import com.yoctopuce.YoctoAPI.YLed;
py	from yocto_led import *
php	require_once('yocto_led.php');
es	in HTML: <script src="../../lib/yocto_led.js"></script> in node.js: require('yoctolib-es2017/yocto_led.js');

Global functions
yFindLed(**func**) Retrieves a LED for a given identifier.
yFindLedInContext(**yctx**, **func**) Retrieves a LED for a given identifier in a YAPI context.
yFirstLed() Starts the enumeration of LEDs currently accessible.
yFirstLedInContext(**yctx**) Starts the enumeration of LEDs currently accessible.
YLed methods
led→clearCache() Invalidates the cache.
led→describe() Returns a short text that describes unambiguously the instance of the LED in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
led→get_advertisedValue() Returns the current value of the LED (no more than 6 characters).
led→get_blinking() Returns the current LED signaling mode.
led→get_errorMessage() Returns the error message of the latest error with the LED.
led→get_errorType() Returns the numerical error code of the latest error with the LED.
led→get_friendlyName() Returns a global identifier of the LED in the format `MODULE_NAME.FUNCTION_NAME`.
led→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
led→get_functionId() Returns the hardware identifier of the LED, without reference to the module.

led→get_hardwareId()

Returns the unique hardware identifier of the LED in the form `SERIAL.FUNCTIONID`.

led→get_logicalName()

Returns the logical name of the LED.

led→get_luminosity()

Returns the current LED intensity (in per cent).

led→get_module()

Gets the `YModule` object for the device on which the function is located.

led→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

led→get_power()

Returns the current LED state.

led→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

led→isOnline()

Checks if the LED is currently reachable, without raising any error.

led→isOnline_async(callback, context)

Checks if the LED is currently reachable, without raising any error (asynchronous version).

led→load(msValidity)

Preloads the LED cache with a specified validity duration.

led→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

led→load_async(msValidity, callback, context)

Preloads the LED cache with a specified validity duration (asynchronous version).

led→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

led→nextLed()

Continues the enumeration of LEDs started using `yFirstLed()`.

led→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

led→set_blinking(newval)

Changes the current LED signaling mode.

led→set_logicalName(newval)

Changes the logical name of the LED.

led→set_luminosity(newval)

Changes the current LED intensity (in per cent).

led→set_power(newval)

Changes the state of the LED.

led→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

led→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

led→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLed.FindLed() YLed
yFindLed()yFindLed()

Retrieves a LED for a given identifier.

```
function FindLed( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the LED is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLed.isOnline()` to test if the LED is indeed online at a given time. In case of ambiguity when looking for a LED by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the LED

Returns :

a `YLed` object allowing you to drive the LED.

YLed.FindLedInContext() YLed yFindLedInContext()

Retrieves a LED for a given identifier in a YAPI context.

function **FindLedInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the LED

Returns :

a `YLed` object allowing you to drive the LED.

YLed.FirstLed() YLed yFirstLed()yFirstLed()

Starts the enumeration of LEDs currently accessible.

function **FirstLed**( )

Use the method `YLed.nextLed()` to iterate on next LEDs.

Returns :

a pointer to a `YLed` object, corresponding to the first LED currently online, or a `null` pointer if there are none.

YLed.FirstLedInContext() YLed yFirstLedInContext()

Starts the enumeration of LEDs currently accessible.

function **FirstLedInContext**(**yctx**)

Use the method `YLed.nextLed()` to iterate on next LEDs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YLed` object, corresponding to the first LED currently online, or a `null` pointer if there are none.

led→clearCache()led.clearCache() YLed

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the LED attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

led→describe()led.describe() YLed

Returns a short text that describes unambiguously the instance of the LED in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the LED (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

led→get_advertisedValue() YLed
led→advertisedValue()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() YLed
led→blinking()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() YLed
led→errorMessage()led.get_errorMessage()

Returns the error message of the latest error with the LED.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the LED object

led→get_errorType() YLed
led→errorType()led.get_errorType()

Returns the numerical error code of the latest error with the LED.

```
function get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the LED object

led→get_friendlyName() YLed
led→friendlyName()led.get_friendlyName()

Returns a global identifier of the LED in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the LED if they are defined, otherwise the serial number of the module and the hardware identifier of the LED (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the LED using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

led→get_functionDescriptor() YLed
led→functionDescriptor()
led.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

led→get_functionId() YLed
led→functionId()led.get_functionId()

Returns the hardware identifier of the LED, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the LED (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

led→get_hardwareId() YLed
led→hardwareId()led.get_hardwareId()

Returns the unique hardware identifier of the LED in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the LED (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the LED (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

led→get_logicalName() YLed
led→logicalName()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() YLed
led→luminosity()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() YLed
led→module()led.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

led→get_power() YLed
led→power()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() YLed
led→userData()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()led.isOnline() YLed

Checks if the LED is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the LED in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the LED.

Returns :

`true` if the LED can be reached, and `false` otherwise

led→load()led.load() YLed

Preloads the LED cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()led.loadAttribute() YLed

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

led→muteValueCallbacks()led.muteValueCallbacks() YLed

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

led→nextLed()led.nextLed() YLed

Continues the enumeration of LEDs started using `yFirstLed()`.

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() YLed
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() YLed
led→setBlinking()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() YLed
led→setLogicalName()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() YLed
led→setLuminosity()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() YLed
led→setPower()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() YLed
led→setUserData()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

led→unmuteValueCallbacks() YLed
led.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

led→wait_async()led.wait_async() YLed

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.35. LightSensor function interface

The Yoctopuce class YLightSensor allows you to read and configure Yoctopuce light sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to easily perform a one-point linear calibration to compensate the effect of a glass or filter placed in front of the sensor. For some light sensors with several working modes, this class can select the desired working mode.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_lightsensor.js'></script>
cpp	#include "yocto_lightsensor.h"
m	#import "yocto_lightsensor.h"
pas	uses yocto_lightsensor;
vb	yocto_lightsensor.vb
cs	yocto_lightsensor.cs
java	import com.yoctopuce.YoctoAPI.YLightSensor;
uwp	import com.yoctopuce.YoctoAPI.YLightSensor;
py	from yocto_lightsensor import *
php	require_once('yocto_lightsensor.php');
es	in HTML: <script src="../../lib/yocto_lightsensor.js"></script> in node.js: require('yoctolib-es2017/yocto_lightsensor.js');

Global functions
yFindLightSensor(**func**) Retrieves a light sensor for a given identifier.
yFindLightSensorInContext(**yctx**, **func**) Retrieves a light sensor for a given identifier in a YAPI context.
yFirstLightSensor() Starts the enumeration of light sensors currently accessible.
yFirstLightSensorInContext(**yctx**) 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→clearCache() Invalidates the cache.
lightsensor→describe() Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
lightsensor→get_advertisedValue() Returns the current value of the light sensor (no more than 6 characters).
lightsensor→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.
lightsensor→get_currentValue()

Returns the current value of the ambient light, in the specified unit, as a floating point number.

lightsensor→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

lightsensor→get_errorMessage()

Returns the error message of the latest error with the light sensor.

lightsensor→get_errorType()

Returns the numerical error code of the latest error with the light sensor.

lightsensor→get_friendlyName()

Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

lightsensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

lightsensor→get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

lightsensor→get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

lightsensor→get_highestValue()

Returns the maximal value observed for the ambient light since the device was started.

lightsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

lightsensor→get_logicalName()

Returns the logical name of the light sensor.

lightsensor→get_lowestValue()

Returns the minimal value observed for the ambient light since the device was started.

lightsensor→get_measureType()

Returns the type of light measure.

lightsensor→get_module()

Gets the `YModule` object for the device on which the function is located.

lightsensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

lightsensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

lightsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

lightsensor→get_resolution()

Returns the resolution of the measured values.

lightsensor→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

lightsensor→get_unit()

Returns the measuring unit for the ambient light.

lightsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

lightsensor→isOnline()

Checks if the light sensor is currently reachable, without raising any error.

lightsensor→isOnline_async(callback, context)

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

lightsensor→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

lightsensor→load(msValidity)

Preloads the light sensor cache with a specified validity duration.

lightsensor→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

lightsensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

lightsensor→load_async(msValidity, callback, context)

Preloads the light sensor cache with a specified validity duration (asynchronous version).

lightsensor→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

lightsensor→nextLightSensor()

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

lightsensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

lightsensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

lightsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

lightsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

lightsensor→set_logicalName(newval)

Changes the logical name of the light sensor.

lightsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

lightsensor→set_measureType(newval)

Modifies the light sensor type used in the device.

lightsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

lightsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

lightsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

lightsensor→startDataLogger()

Starts the data logger on the device.

lightsensor→stopDataLogger()

Stops the datalogger on the device.

lightsensor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

lightsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLightSensor.FindLightSensor() YLightSensor yFindLightSensor()yFindLightSensor()

Retrieves a light sensor for a given identifier.

function **FindLightSensor**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the light sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLightSensor.isOnline()` to test if the light sensor is indeed online at a given time. In case of ambiguity when looking for a light sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the light sensor

Returns :

a `YLightSensor` object allowing you to drive the light sensor.

YLightSensor.FindLightSensorInContext() YLightSensor yFindLightSensorInContext()

Retrieves a light sensor for a given identifier in a YAPI context.

function **FindLightSensorInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the light sensor

Returns :

a `YLightSensor` object allowing you to drive the light sensor.

YLightSensor.FirstLightSensor() YLightSensor yFirstLightSensor()yFirstLightSensor()

Starts the enumeration of light sensors currently accessible.

function **FirstLightSensor**( )

Use the method `YLightSensor.nextLightSensor()` to iterate on next light sensors.

Returns :

a pointer to a `YLightSensor` object, corresponding to the first light sensor currently online, or a `null` pointer if there are none.

YLightSensor.FirstLightSensorInContext() YLightSensor yFirstLightSensorInContext()

Starts the enumeration of light sensors currently accessible.

function **FirstLightSensorInContext**(**yctx**)

Use the method `YLightSensor.nextLightSensor()` to iterate on next light sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YLightSensor` object, corresponding to the first light sensor currently online, or a `null` pointer if there are none.

lightsensor→calibrate()lightsensor.calibrate() YLightSensor

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

function **calibrate**(**calibratedVal**)

Parameters :

calibratedVal the desired target value.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→calibrateFromPoints() YLightSensor
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.

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→clearCache()lightsensor.clearCache() YLightSensor

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the light sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

lightsensor→describe()lightsensor.describe() YLightSensor

Returns a short text that describes unambiguously the instance of the light sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the light sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

lightsensor→get_advertisedValue() YLightSensor
lightsensor→advertisedValue()
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() YLightSensor
lightsensor→currentRawValue()
lightsensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

lightsensor→get_currentValue() YLightSensor
lightsensor→currentValue()
lightsensor.get_currentValue()

Returns the current value of the ambient light, in the specified unit, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the ambient light, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

lightsensor→get_dataLogger() YLightSensor
lightsensor→dataLogger()
lightsensor.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

lightsensor→get_errorMessage() YLightSensor
lightsensor→errorMessage()
lightsensor.get_errorMessage()

Returns the error message of the latest error with the light sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the light sensor object

lightsensor→get_errorType() YLightSensor
lightsensor→errorType()lightsensor.get_errorType()

Returns the numerical error code of the latest error with the light sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the light sensor object

lightsensor→get_friendlyName() YLightSensor
lightsensor→friendlyName()
lightsensor.get_friendlyName()

Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the light sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the light sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the light sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

lightsensor→get_functionDescriptor() YLightSensor
lightsensor→functionDescriptor()
lightsensor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

lightsensor→get_functionId() YLightSensor
lightsensor→functionId()lightsensor.get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the light sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

lightsensor→get_hardwareId() YLightSensor
lightsensor→hardwareId()
lightsensor.get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the light sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the light sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

lightsensor→get_highestValue() YLightSensor
lightsensor→highestValue()
lightsensor.get_highestValue()

Returns the maximal value observed for the ambient light since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the ambient light since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

lightsensor→get_logFrequency() YLightSensor
lightsensor→logFrequency()
lightsensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

lightsensor→get_logicalName() YLightSensor
lightsensor→logicalName()
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() YLightSensor
lightsensor→lowestValue()
lightsensor.get_lowestValue()

Returns the minimal value observed for the ambient light since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the ambient light since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

lightsensor→get_measureType() YLightSensor
lightsensor→measureType()
lightsensor.get_measureType()

Returns the type of light measure.

function **get_measureType**( )

Returns :

a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY` corresponding to the type of light measure

On failure, throws an exception or returns `Y_MEASURETYPE_INVALID`.

lightsensor→get_module() YLightSensor
lightsensor→module()lightsensor.get_module()

Gets the YModule object for the device on which the function is located.

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_recordedData() YLightSensor
lightsensor→recordedData()
lightsensor.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

lightsensor→get_reportFrequency() YLightSensor
lightsensor→reportFrequency()
lightsensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

lightsensor→get_resolution() YLightSensor
lightsensor→resolution()lightsensor.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

lightsensor→get_sensorState() YLightSensor
lightsensor→sensorState()
lightsensor.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :
an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

lightsensor→get_unit() YLightSensor
lightsensor→unit()lightsensor.get_unit()

Returns the measuring unit for the ambient light.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the ambient light

On failure, throws an exception or returns `Y_UNIT_INVALID`.

lightsensor→get_userData() YLightSensor lightsensor→userData()lightsensor.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

lightsensor→isOnline()lightsensor.isOnline() YLightSensor

Checks if the light sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

Returns :

`true` if the light sensor can be reached, and `false` otherwise

lightsensor→load()lightsensor.load() YLightSensor

Preloads the light sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute() YLightSensor
lightsensor.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

lightsensor→loadCalibrationPoints() YLightSensor
lightsensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→muteValueCallbacks() YLightSensor
lightsensor.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→nextLightSensor() YLightSensor
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→registerTimedReportCallback() YLightSensor
lightsensor.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

lightsensor→registerValueCallback() YLightSensor
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() YLightSensor
lightsensor→setHighestValue()
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_logFrequency() YLightSensor
lightsensor→setLogFrequency()
lightsensor.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logicalName() YLightSensor
lightsensor→setLogicalName()
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() YLightSensor
lightsensor→setLowestValue()
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_measureType() YLightSensor
lightsensor→setMeasureType()
lightsensor.set_measureType()

Modifies the light sensor type used in the device.

function **set_measureType**(**newval**)

The measure can either approximate the response of the human eye, focus on a specific light spectrum, depending on the capabilities of the light-sensitive cell. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_reportFrequency() YLightSensor
lightsensor→setReportFrequency()
lightsensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_resolution() YLightSensor
lightsensor→setResolution()
lightsensor.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_userData() YLightSensor
lightsensor→setUserData()
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

lightsensor→startDataLogger() YLightSensor
lightsensor.startDataLogger()

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

lightsensor→stopDataLogger() YLightSensor
lightsensor.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

lightsensor→unmuteValueCallbacks() YLightSensor
lightsensor.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→wait_async()lightsensor.wait_async() YLightSensor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.36. Longitude function interface

The Yoctopuce class YLongitude allows you to read the longitude from Yoctopuce geolocalization sensors. It inherits from the YSensor class the core functions to read measurements, register callback functions, access the autonomous datalogger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_longitude.js'></script>
cpp	#include "yocto_longitude.h"
m	#import "yocto_longitude.h"
pas	uses yocto_longitude;
vb	yocto_longitude.vb
cs	yocto_longitude.cs
java	import com.yoctopuce.YoctoAPI.YLongitude;
uwp	import com.yoctopuce.YoctoAPI.YLongitude;
py	from yocto_longitude import *
php	require_once('yocto_longitude.php');
es	in HTML: <script src="../../lib/yocto_longitude.js"></script> in node.js: require('yoctolib-es2017/yocto_longitude.js');

Global functions	
yFindLongitude(**func**)	Retrieves a longitude sensor for a given identifier.
yFindLongitudeInContext(**yctx**, **func**)	Retrieves a longitude sensor for a given identifier in a YAPI context.
yFirstLongitude()	Starts the enumeration of longitude sensors currently accessible.
yFirstLongitudeInContext(**yctx**)	Starts the enumeration of longitude sensors currently accessible.
`YLongitude` methods	
longitude→calibrateFromPoints(**rawValues**, **refValues**)	Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
longitude→clearCache()	Invalidates the cache.
longitude→describe()	Returns a short text that describes unambiguously the instance of the longitude sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
longitude→get_advertisedValue()	Returns the current value of the longitude sensor (no more than 6 characters).
longitude→get_currentRawValue()	Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.
longitude→get_currentValue()	Returns the current value of the longitude, in deg/1000, as a floating point number.
longitude→get_dataLogger()	Returns the YDatalogger object of the device hosting the sensor.
longitude→get_errorMessage()	Returns the error message of the latest error with the longitude sensor.

longitude→get_errorType()

Returns the numerical error code of the latest error with the longitude sensor.

longitude→get_friendlyName()

Returns a global identifier of the longitude sensor in the format `MODULE_NAME.FUNCTION_NAME`.

longitude→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

longitude→get_functionId()

Returns the hardware identifier of the longitude sensor, without reference to the module.

longitude→get_hardwareId()

Returns the unique hardware identifier of the longitude sensor in the form `SERIAL.FUNCTIONID`.

longitude→get_highestValue()

Returns the maximal value observed for the longitude since the device was started.

longitude→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

longitude→get_logicalName()

Returns the logical name of the longitude sensor.

longitude→get_lowestValue()

Returns the minimal value observed for the longitude since the device was started.

longitude→get_module()

Gets the `YModule` object for the device on which the function is located.

longitude→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

longitude→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

longitude→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

longitude→get_resolution()

Returns the resolution of the measured values.

longitude→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

longitude→get_unit()

Returns the measuring unit for the longitude.

longitude→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

longitude→isOnline()

Checks if the longitude sensor is currently reachable, without raising any error.

longitude→isOnline_async(**callback**, **context**)

Checks if the longitude sensor is currently reachable, without raising any error (asynchronous version).

longitude→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

longitude→load(**msValidity**)

Preloads the longitude sensor cache with a specified validity duration.

longitude→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

longitude→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

longitude→load_async(msValidity, callback, context)

Preloads the longitude sensor cache with a specified validity duration (asynchronous version).

longitude→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

longitude→nextLongitude()

Continues the enumeration of longitude sensors started using `yFirstLongitude()`.

longitude→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

longitude→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

longitude→set_highestValue(newval)

Changes the recorded maximal value observed.

longitude→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

longitude→set_logicalName(newval)

Changes the logical name of the longitude sensor.

longitude→set_lowestValue(newval)

Changes the recorded minimal value observed.

longitude→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

longitude→set_resolution(newval)

Changes the resolution of the measured physical values.

longitude→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

longitude→startDataLogger()

Starts the data logger on the device.

longitude→stopDataLogger()

Stops the datalogger on the device.

longitude→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

longitude→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLongitude.FindLongitude() YLongitude
yFindLongitude()yFindLongitude()

Retrieves a longitude sensor for a given identifier.

```
function FindLongitude( 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 longitude sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLongitude.isOnline()` to test if the longitude sensor is indeed online at a given time. In case of ambiguity when looking for a longitude 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the longitude sensor

Returns :

a `YLongitude` object allowing you to drive the longitude sensor.

YLongitude.FindLongitudeInContext() YLongitude yFindLongitudeInContext()

Retrieves a longitude sensor for a given identifier in a YAPI context.

function **FindLongitudeInContext**(**yctx**, **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 longitude sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLongitude.isOnline()` to test if the longitude sensor is indeed online at a given time. In case of ambiguity when looking for a longitude 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 :

yctx a YAPI context

func a string that uniquely characterizes the longitude sensor

Returns :

a `YLongitude` object allowing you to drive the longitude sensor.

YLongitude.FirstLongitude() YLongitude yFirstLongitude()yFirstLongitude()

Starts the enumeration of longitude sensors currently accessible.

function **FirstLongitude**( )

Use the method `YLongitude.nextLongitude()` to iterate on next longitude sensors.

Returns :

a pointer to a `YLongitude` object, corresponding to the first longitude sensor currently online, or a `null` pointer if there are none.

YLongitude.FirstLongitudeInContext() YLongitude yFirstLongitudeInContext()

Starts the enumeration of longitude sensors currently accessible.

function **FirstLongitudeInContext**(**yctx**)

Use the method `YLongitude.nextLongitude()` to iterate on next longitude sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YLongitude` object, corresponding to the first longitude sensor currently online, or a `null` pointer if there are none.

longitude→calibrateFromPoints() YLongitude
longitude.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.

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.

longitude→clearCache()longitude.clearCache() YLongitude

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the longitude sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

longitude→describe()longitude.describe() YLongitude

Returns a short text that describes unambiguously the instance of the longitude sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the longitude sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

longitude→get_advertisedValue() YLongitude
longitude→advertisedValue()
longitude.get_advertisedValue()

Returns the current value of the longitude sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the longitude sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

longitude→get_currentRawValue() YLongitude
longitude→currentRawValue()
longitude.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

longitude→get_currentValue() YLongitude
longitude→currentValue()
longitude.get_currentValue()

Returns the current value of the longitude, in deg/1000, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the longitude, in deg/1000, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

longitude→get_dataLogger() YLongitude
longitude→dataLogger()longitude.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

longitude→get_errorMessage() YLongitude
longitude→errorMessage()
longitude.get_errorMessage()

Returns the error message of the latest error with the longitude sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the longitude sensor object

longitude→get_errorType() YLongitude
longitude→errorType()longitude.get_errorType()

Returns the numerical error code of the latest error with the longitude sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the longitude sensor object

longitude→get_friendlyName() YLongitude
longitude→friendlyName()
longitude.get_friendlyName()

Returns a global identifier of the longitude sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the longitude sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the longitude sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the longitude sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

longitude→get_functionDescriptor() YLongitude
longitude→functionDescriptor()
longitude.get_functionDescriptor()

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.

longitude→get_functionId() YLongitude
longitude→functionId()longitude.get_functionId()

Returns the hardware identifier of the longitude sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the longitude sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

longitude→get_hardwareId() YLongitude
longitude→hardwareId()longitude.get_hardwareId()

Returns the unique hardware identifier of the longitude sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the longitude sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the longitude sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

longitude→get_highestValue() YLongitude
longitude→highestValue()
longitude.get_highestValue()

Returns the maximal value observed for the longitude since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the longitude since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

longitude→get_logFrequency() YLongitude
longitude→logFrequency()
longitude.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

longitude→get_logicalName() YLongitude
longitude→logicalName()
longitude.get_logicalName()

Returns the logical name of the longitude sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the longitude sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

longitude→get_lowestValue() YLongitude
longitude→lowestValue()longitude.get_lowestValue()

Returns the minimal value observed for the longitude since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the longitude since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

longitude→get_module() YLongitude
longitude→module()longitude.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

longitude→get_recordedData() YLongitude
longitude→recordedData()
longitude.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

longitude→get_reportFrequency() YLongitude
longitude→reportFrequency()
longitude.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

longitude→get_resolution() YLongitude
longitude→resolution()longitude.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

longitude→get_sensorState() YLongitude
longitude→sensorState()longitude.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

longitude→get_unit() YLongitude
longitude→unit()longitude.get_unit()

Returns the measuring unit for the longitude.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the longitude

On failure, throws an exception or returns `Y_UNIT_INVALID`.

longitude→get_userData() YLongitude
longitude→userData()longitude.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.

longitude→isOnline()longitude.isOnline() YLongitude

Checks if the longitude sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the longitude sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the longitude sensor.

Returns :

`true` if the longitude sensor can be reached, and `false` otherwise

longitude→load()longitude.load() YLongitude

Preloads the longitude sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

longitude→loadAttribute()longitude.loadAttribute() YLongitude

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

longitude→loadCalibrationPoints() YLongitude
longitude.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→muteValueCallbacks() YLongitude
longitude.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→nextLongitude()longitude.nextLongitude() YLongitude

Continues the enumeration of longitude sensors started using `yFirstLongitude()`.

function **nextLongitude**( )

Returns :

a pointer to a `YLongitude` object, corresponding to a longitude sensor currently online, or a `null` pointer if there are no more longitude sensors to enumerate.

longitude→registerTimedReportCallback() YLongitude
longitude.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

longitude→registerValueCallback() YLongitude
longitude.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.

longitude→set_highestValue() YLongitude
longitude→setHighestValue()
longitude.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.

longitude→set_logFrequency() YLongitude
longitude→setLogFrequency()
longitude.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→set_logicalName() YLongitude
longitude→setLogicalName()
longitude.set_logicalName()

Changes the logical name of the longitude 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 longitude sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→set_lowestValue() YLongitude
longitude→setLowestValue()
longitude.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.

longitude→set_reportFrequency() YLongitude
longitude→setReportFrequency()
longitude.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→set_resolution() YLongitude
longitude→setResolution()longitude.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→set_userData() YLongitude
longitude→setUserData()longitude.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

longitude→startDataLogger() YLongitude
longitude.startDataLogger()

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

longitude→stopDataLogger() YLongitude
longitude.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

longitude→unmuteValueCallbacks() YLongitude
longitude.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

longitude→wait_async()longitude.wait_async() YLongitude

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.37. Magnetometer function interface

The YSensor class is the parent class for all Yoctopuce sensors. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provide a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButton class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_magnetometer.js'></script>
cpp	#include "yocto_magnetometer.h"
m	#import "yocto_magnetometer.h"
pas	uses yocto_magnetometer;
vb	yocto_magnetometer.vb
cs	yocto_magnetometer.cs
java	import com.yoctopuce.YoctoAPI.YMagnetometer;
uwp	import com.yoctopuce.YoctoAPI.YMagnetometer;
py	from yocto_magnetometer import *
php	require_once('yocto_magnetometer.php');
es	in HTML: <script src="../../lib/yocto_magnetometer.js"></script> in node.js: require('yoctolib-es2017/yocto_magnetometer.js');

Global functions
yFindMagnetometer(**func**) Retrieves a magnetometer for a given identifier.
yFindMagnetometerInContext(**yctx**, **func**) Retrieves a magnetometer for a given identifier in a YAPI context.
yFirstMagnetometer() Starts the enumeration of magnetometers currently accessible.
yFirstMagnetometerInContext(**yctx**) Starts the enumeration of magnetometers currently accessible.
`YMagnetometer` methods
magnetometer→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
magnetometer→clearCache() Invalidates the cache.
magnetometer→describe() Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
magnetometer→get_advertisedValue() Returns the current value of the magnetometer (no more than 6 characters).
magnetometer→get_bandwidth() Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).
magnetometer→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number.
magnetometer→get_currentValue() Returns the current value of the magnetic field, in mT, as a floating point number.

magnetometer→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

magnetometer→get_errorMessage()

Returns the error message of the latest error with the magnetometer.

magnetometer→get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

magnetometer→get_friendlyName()

Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.

magnetometer→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

magnetometer→get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

magnetometer→get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form `SERIAL.FUNCTIONID`.

magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

magnetometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

magnetometer→get_logicalName()

Returns the logical name of the magnetometer.

magnetometer→get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

magnetometer→get_module()

Gets the `YModule` object for the device on which the function is located.

magnetometer→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

magnetometer→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

magnetometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

magnetometer→get_resolution()

Returns the resolution of the measured values.

magnetometer→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

magnetometer→get_unit()

Returns the measuring unit for the magnetic field.

magnetometer→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

magnetometer→get_xValue()

Returns the X component of the magnetic field, as a floating point number.

magnetometer→get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

magnetometer→get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

magnetometer→isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

magnetometer→isOnline_async(**callback**, **context**)

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

magnetometer→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

magnetometer→load(**msValidity**)

Preloads the magnetometer cache with a specified validity duration.

magnetometer→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

magnetometer→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

magnetometer→load_async(**msValidity**, **callback**, **context**)

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

magnetometer→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

magnetometer→nextMagnetometer()

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

magnetometer→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

magnetometer→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

magnetometer→set_bandwidth(**newval**)

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

magnetometer→set_highestValue(**newval**)

Changes the recorded maximal value observed.

magnetometer→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

magnetometer→set_logicalName(**newval**)

Changes the logical name of the magnetometer.

magnetometer→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

magnetometer→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

magnetometer→set_resolution(**newval**)

Changes the resolution of the measured physical values.

magnetometer→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

magnetometer→startDataLogger()

Starts the data logger on the device.

magnetometer→stopDataLogger()

Stops the datalogger on the device.

magnetometer→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

magnetometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMagnetometer.FindMagnetometer() YMagnetometer yFindMagnetometer()yFindMagnetometer()

Retrieves a magnetometer for a given identifier.

```
function FindMagnetometer( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMagnetometer.isOnline()` to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the magnetometer

Returns :

a `YMagnetometer` object allowing you to drive the magnetometer.

YMagnetometer.FindMagnetometerInContext() YMagnetometer yFindMagnetometerInContext()

Retrieves a magnetometer for a given identifier in a YAPI context.

function **FindMagnetometerInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMagnetometer.isOnline()` to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the magnetometer

Returns :

a `YMagnetometer` object allowing you to drive the magnetometer.

YMagnetometer.FirstMagnetometer() YMagnetometer yFirstMagnetometer()yFirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

function **FirstMagnetometer**( )

Use the method `YMagnetometer.nextMagnetometer()` to iterate on next magnetometers.

Returns :

a pointer to a `YMagnetometer` object, corresponding to the first magnetometer currently online, or a `null` pointer if there are none.

YMagnetometer.FirstMagnetometerInContext() YMagnetometer yFirstMagnetometerInContext()

Starts the enumeration of magnetometers currently accessible.

function **FirstMagnetometerInContext**(**yctx**)

Use the method `YMagnetometer.nextMagnetometer()` to iterate on next magnetometers.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YMagnetometer` object, corresponding to the first magnetometer currently online, or a `null` pointer if there are none.

magnetometer→calibrateFromPoints() YMagnetometer
magnetometer.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→clearCache() YMagnetometer
magnetometer.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the magnetometer attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

magnetometer→describe()magnetometer.describe() YMagnetometer

Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the magnetometer (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

magnetometer→get_advertisedValue() YMagnetometer
magnetometer→advertisedValue()
magnetometer.get_advertisedValue()

Returns the current value of the magnetometer (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the magnetometer (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

magnetometer→get_bandwidth() YMagnetometer
magnetometer→bandwidth()
magnetometer.get_bandwidth()

Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **get_bandwidth**( )

Returns :
an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

On failure, throws an exception or returns `Y_BANDWIDTH_INVALID`.

magnetometer→get_currentRawValue() YMagnetometer
magnetometer→currentRawValue()
magnetometer.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number.

function **get_currentRawValue**( )

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

magnetometer→get_currentValue() YMagnetometer
magnetometer→currentValue()
magnetometer.get_currentValue()

Returns the current value of the magnetic field, in mT, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the magnetic field, in mT, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

magnetometer→get_dataLogger() YMagnetometer
magnetometer→dataLogger()
magnetometer.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

magnetometer→get_errorMessage() YMagnetometer
magnetometer→errorMessage()
magnetometer.get_errorMessage()

Returns the error message of the latest error with the magnetometer.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the magnetometer object

magnetometer→get_errorType() YMagnetometer
magnetometer→errorType()
magnetometer.get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the magnetometer object

magnetometer→get_friendlyName() YMagnetometer
magnetometer→friendlyName()
magnetometer.get_friendlyName()

Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the magnetometer if they are defined, otherwise the serial number of the module and the hardware identifier of the magnetometer (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the magnetometer using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

magnetometer→get_functionDescriptor() YMagnetometer
magnetometer→functionDescriptor()
magnetometer.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

magnetometer→get_functionId() YMagnetometer
magnetometer→functionId()
magnetometer.get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the magnetometer (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

magnetometer→get_hardwareId() YMagnetometer
magnetometer→hardwareId()
magnetometer.get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the magnetometer (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the magnetometer (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

magnetometer→get_highestValue() YMagnetometer
magnetometer→highestValue()
magnetometer.get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

magnetometer→get_logFrequency() YMagnetometer
magnetometer→logFrequency()
magnetometer.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

magnetometer→get_logicalName() YMagnetometer
magnetometer→logicalName()
magnetometer.get_logicalName()

Returns the logical name of the magnetometer.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the magnetometer.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

magnetometer→get_lowestValue() YMagnetometer
magnetometer→lowestValue()
magnetometer.get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

magnetometer→get_module() YMagnetometer
magnetometer→module()
magnetometer.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

magnetometer→get_recordedData() YMagnetometer
magnetometer→recordedData()
magnetometer.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

magnetometer→get_reportFrequency() YMagnetometer
magnetometer→reportFrequency()
magnetometer.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

magnetometer→get_resolution() YMagnetometer
magnetometer→resolution()
magnetometer.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

magnetometer→get_sensorState() YMagnetometer
magnetometer→sensorState()
magnetometer.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

magnetometer→get_unit() YMagnetometer
magnetometer→unit()magnetometer.get_unit()

Returns the measuring unit for the magnetic field.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns `Y_UNIT_INVALID`.

magnetometer→get_userData() YMagnetometer
magnetometer→userData()
magnetometer.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.

magnetometer→get_xValue() YMagnetometer
magnetometer→xValue()magnetometer.get_xValue()

Returns the X component of the magnetic field, as a floating point number.

function **get_xValue**( )

Returns :
a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

magnetometer→get_yValue() YMagnetometer
magnetometer→yValue()magnetometer.get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

function **get_yValue**( )

Returns :
a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

magnetometer→get_zValue() YMagnetometer
magnetometer→zValue()magnetometer.get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

function **get_zValue**( )

Returns :
a floating point number corresponding to the Z component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_ZVALUE_INVALID.

magnetometer→isOnline()magnetometer.isOnline() YMagnetometer

Checks if the magnetometer is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the magnetometer.

Returns :

`true` if the magnetometer can be reached, and `false` otherwise

magnetometer→load()magnetometer.load() YMagnetometer

Preloads the magnetometer cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→loadAttribute() YMagnetometer
magnetometer.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

magnetometer→loadCalibrationPoints() YMagnetometer
magnetometer.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→muteValueCallbacks() YMagnetometer
magnetometer.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→nextMagnetometer() YMagnetometer
magnetometer.nextMagnetometer()

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

function **nextMagnetometer**( )

Returns :

a pointer to a `YMagnetometer` object, corresponding to a magnetometer currently online, or a `null` pointer if there are no more magnetometers to enumerate.

magnetometer→registerTimedReportCallback() YMagnetometer
magnetometer.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

magnetometer→registerValueCallback() YMagnetometer
magnetometer.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.

magnetometer→set_bandwidth() YMagnetometer
magnetometer→setBandwidth()
magnetometer.set_bandwidth()

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

```
function set_bandwidth( newval)
```

When the frequency is lower, the device performs averaging.

Parameters :

newval an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_highestValue() YMagnetometer
magnetometer→setHighestValue()
magnetometer.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.

magnetometer→set_logFrequency() YMagnetometer
magnetometer→setLogFrequency()
magnetometer.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logicalName() YMagnetometer
magnetometer→setLogicalName()
magnetometer.set_logicalName()

Changes the logical name of the magnetometer.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the magnetometer.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_lowestValue() YMagnetometer
magnetometer→setLowestValue()
magnetometer.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.

magnetometer→set_reportFrequency() YMagnetometer
magnetometer→setReportFrequency()
magnetometer.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_resolution() YMagnetometer
magnetometer→setResolution()
magnetometer.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_userData() YMagnetometer
magnetometer→setUserData()
magnetometer.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

magnetometer→startDataLogger() YMagnetometer
magnetometer.startDataLogger()

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

magnetometer→stopDataLogger() YMagnetometer
magnetometer.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

magnetometer→unmuteValueCallbacks() YMagnetometer
magnetometer.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→wait_async() YMagnetometer
magnetometer.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.38. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');

YMeasure methods
measure→get_averageValue() Returns the average value observed during the time interval covered by this measure.
measure→get_endTimeUTC() Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).
measure→get_maxValue() Returns the largest value observed during the time interval covered by this measure.
measure→get_minValue() Returns the smallest value observed during the time interval covered by this measure.
measure→get_startTimeUTC() Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_averageValue() YMeasure
measure→averageValue()
measure.get_averageValue()

Returns the average value observed during the time interval covered by this measure.

function **get_averageValue**( )

Returns :
a floating-point number corresponding to the average value observed.

measure→get_endTimeUTC() YMeasure
measure→endTimeUTC()measure.get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

function **get_endTimeUTC**( )

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→get_maxValue() YMeasure
measure→maxValue()measure.get_maxValue()

Returns the largest value observed during the time interval covered by this measure.

function **get_maxValue**( )

Returns :
a floating-point number corresponding to the largest value observed.

measure→get_minValue() YMeasure
measure→minValue()measure.get_minValue()

Returns the smallest value observed during the time interval covered by this measure.

function **get_minValue**( )

Returns :
a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC() YMeasure
measure→startTimeUTC()
measure.get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

function **get_startTimeUTC**( )

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :
an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

3.39. MessageBox function interface

YMessageBox functions provides SMS sending and receiving capability to GSM-enabled Yoctopuce devices.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_messagebox.js'></script>
cpp	#include "yocto_messagebox.h"
m	#import "yocto_messagebox.h"
pas	uses yocto_messagebox;
vb	yocto_messagebox.vb
cs	yocto_messagebox.cs
java	import com.yoctopuce.YoctoAPI.YMessageBox;
uwp	import com.yoctopuce.YoctoAPI.YMessageBox;
py	from yocto_messagebox import *
php	require_once('yocto_messagebox.php');
es	in HTML: <script src="../../lib/yocto_messagebox.js"></script> in node.js: require('yoctolib-es2017/yocto_messagebox.js');

Global functions

yFindMessageBox(**func**)

Retrieves a MessageBox interface for a given identifier.

yFindMessageBoxInContext(**yctx**, **func**)

Retrieves a MessageBox interface for a given identifier in a YAPI context.

yFirstMessageBox()

Starts the enumeration of MessageBox interfaces currently accessible.

yFirstMessageBoxInContext(**yctx**)

Starts the enumeration of MessageBox interfaces currently accessible.

YMessageBox methods

messagebox→clearCache()

Invalidates the cache.

messagebox→clearPduCounters()

Clear the SMS units counters.

messagebox→describe()

Returns a short text that describes unambiguously the instance of the MessageBox interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

messagebox→get_advertisedValue()

Returns the current value of the MessageBox interface (no more than 6 characters).

messagebox→get_errorMessage()

Returns the error message of the latest error with the MessageBox interface.

messagebox→get_errorType()

Returns the numerical error code of the latest error with the MessageBox interface.

messagebox→get_friendlyName()

Returns a global identifier of the MessageBox interface in the format `MODULE_NAME.FUNCTION_NAME`.

messagebox→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

messagebox→get_functionId()

Returns the hardware identifier of the MessageBox interface, without reference to the module.

messagebox→get_hardwareId()

Returns the unique hardware identifier of the MessageBox interface in the form `SERIAL.FUNCTIONID`.

messagebox→get_logicalName()

Returns the logical name of the MessageBox interface.

messagebox→get_messages()

Returns the list of messages received and not deleted.

messagebox→get_module()

Gets the `YModule` object for the device on which the function is located.

messagebox→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

messagebox→get_pduReceived()

Returns the number of SMS units received so far.

messagebox→get_pduSent()

Returns the number of SMS units sent so far.

messagebox→get_slotsCount()

Returns the total number of message storage slots on the SIM card.

messagebox→get_slotsInUse()

Returns the number of message storage slots currently in use.

messagebox→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

messagebox→isOnline()

Checks if the MessageBox interface is currently reachable, without raising any error.

messagebox→isOnline_async(**callback**, **context**)

Checks if the MessageBox interface is currently reachable, without raising any error (asynchronous version).

messagebox→load(**msValidity**)

Preloads the MessageBox interface cache with a specified validity duration.

messagebox→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

messagebox→load_async(**msValidity**, **callback**, **context**)

Preloads the MessageBox interface cache with a specified validity duration (asynchronous version).

messagebox→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

messagebox→newMessage(**recipient**)

Creates a new empty SMS message, to be configured and sent later on.

messagebox→nextMessageBox()

Continues the enumeration of MessageBox interfaces started using `yFirstMessageBox()`.

messagebox→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

messagebox→sendFlashMessage(**recipient**, **message**)

Sends a Flash SMS (class 0 message).

messagebox→sendTextMessage(**recipient**, **message**)

Sends a regular text SMS, with standard parameters.

messagebox→set_logicalName(**newval**)

Changes the logical name of the MessageBox interface.

messagebox→set_pduReceived(**newval**)

Changes the value of the incoming SMS units counter.

messagebox→set_pduSent(newval)

Changes the value of the outgoing SMS units counter.

messagebox→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

messagebox→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

messagebox→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMessageBox.FindMessageBox() YMessageBox yFindMessageBox()yFindMessageBox()

Retrieves a MessageBox interface for a given identifier.

function **FindMessageBox**(**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 MessageBox interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMessageBox.isOnline()` to test if the MessageBox interface is indeed online at a given time. In case of ambiguity when looking for a MessageBox 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the MessageBox interface

Returns :

a `YMessageBox` object allowing you to drive the MessageBox interface.

YMessageBox.FindMessageBoxInContext() YMessageBox yFindMessageBoxInContext()

Retrieves a MessageBox interface for a given identifier in a YAPI context.

function **FindMessageBoxInContext**(**yctx**, **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 MessageBox interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMessageBox.isOnline()` to test if the MessageBox interface is indeed online at a given time. In case of ambiguity when looking for a MessageBox 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 :

yctx a YAPI context

func a string that uniquely characterizes the MessageBox interface

Returns :

a `YMessageBox` object allowing you to drive the MessageBox interface.

YMessageBox.FirstMessageBox() YMessageBox yFirstMessageBox()yFirstMessageBox()

Starts the enumeration of MessageBox interfaces currently accessible.

function **FirstMessageBox**( )

Use the method `YMessageBox.nextMessageBox()` to iterate on next MessageBox interfaces.

Returns :

a pointer to a `YMessageBox` object, corresponding to the first MessageBox interface currently online, or a `null` pointer if there are none.

YMessageBox.FirstMessageBoxInContext() YMessageBox yFirstMessageBoxInContext()

Starts the enumeration of MessageBox interfaces currently accessible.

function **FirstMessageBoxInContext**(**yctx**)

Use the method `YMessageBox.nextMessageBox()` to iterate on next MessageBox interfaces.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YMessageBox` object, corresponding to the first MessageBox interface currently online, or a `null` pointer if there are none.

messagebox→clearCache() YMessageBox
messagebox.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the MessageBox interface attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

messagebox→clearPduCounters() YMessageBox
messagebox.clearPduCounters()

Clear the SMS units counters.

function **clearPduCounters**( )

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→describe()messagebox.describe() YMessageBox

Returns a short text that describes unambiguously the instance of the MessageBox interface in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the MessageBox interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

messagebox→get_advertisedValue() YMessageBox
messagebox→advertisedValue()
messagebox.get_advertisedValue()

Returns the current value of the MessageBox interface (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the MessageBox interface (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

messagebox→get_errorMessage() YMessageBox
messagebox→errorMessage()
messagebox.get_errorMessage()

Returns the error message of the latest error with the MessageBox interface.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the MessageBox interface object

messagebox→get_errorType() YMessageBox
messagebox→errorType()
messagebox.get_errorType()

Returns the numerical error code of the latest error with the MessageBox interface.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the MessageBox interface object

messagebox→get_friendlyName() YMessageBox
messagebox→friendlyName()
messagebox.get_friendlyName()

Returns a global identifier of the MessageBox interface in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the MessageBox interface if they are defined, otherwise the serial number of the module and the hardware identifier of the MessageBox interface (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the MessageBox interface using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

messagebox→get_functionDescriptor() YMessageBox
messagebox→functionDescriptor()
messagebox.get_functionDescriptor()

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.

messagebox→get_functionId() YMessageBox
messagebox→functionId()
messagebox.get_functionId()

Returns the hardware identifier of the MessageBox interface, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the MessageBox interface (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

messagebox→get_hardwareId() YMessageBox
messagebox→hardwareId()
messagebox.get_hardwareId()

Returns the unique hardware identifier of the MessageBox interface in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the MessageBox interface (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the MessageBox interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

messagebox→get_logicalName() YMessageBox
messagebox→logicalName()
messagebox.get_logicalName()

Returns the logical name of the MessageBox interface.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the MessageBox interface.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

messagebox→get_messages() YMessageBox
messagebox→messages()
messagebox.get_messages()

Returns the list of messages received and not deleted.

function **get_messages**( )

This function will automatically decode concatenated SMS.

Returns :
an YSms object list.

On failure, throws an exception or returns an empty list.

messagebox→get_module() YMessageBox
messagebox→module()messagebox.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

messagebox→get_pduReceived() YMessageBox
messagebox→pduReceived()
messagebox.get_pduReceived()

Returns the number of SMS units received so far.

function **get_pduReceived**( )

Returns :
an integer corresponding to the number of SMS units received so far

On failure, throws an exception or returns `Y_PDURECEIVED_INVALID`.

messagebox→get_pduSent() YMessageBox
messagebox→pduSent()messagebox.get_pduSent()

Returns the number of SMS units sent so far.

function **get_pduSent**( )

Returns :
an integer corresponding to the number of SMS units sent so far

On failure, throws an exception or returns `Y_PDUSENT_INVALID`.

messagebox→get_slotsCount() YMessageBox
messagebox→slotsCount()
messagebox.get_slotsCount()

Returns the total number of message storage slots on the SIM card.

function **get_slotsCount**( )

Returns :
an integer corresponding to the total number of message storage slots on the SIM card

On failure, throws an exception or returns `Y_SLOTSCOUNT_INVALID`.

messagebox→get_slotsInUse() YMessageBox
messagebox→slotsInUse()
messagebox.get_slotsInUse()

Returns the number of message storage slots currently in use.

function **get_slotsInUse**( )

Returns :
an integer corresponding to the number of message storage slots currently in use

On failure, throws an exception or returns `Y_SLOTSINUSE_INVALID`.

messagebox→get_userData() YMessageBox
messagebox→userData()messagebox.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.

messagebox→isOnline()messagebox.isOnline() YMessageBox

Checks if the MessageBox interface is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the MessageBox interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the MessageBox interface.

Returns :

`true` if the MessageBox interface can be reached, and `false` otherwise

messagebox→load()messagebox.load() YMessageBox

Preloads the MessageBox interface cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

messagebox→loadAttribute() YMessageBox
messagebox.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

messagebox→muteValueCallbacks() YMessageBox
messagebox.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→newMessage() YMessageBox
messagebox.newMessage()

Creates a new empty SMS message, to be configured and sent later on.

function **newMessage**(**recipient**)

Parameters :

recipient a text string with the recipient phone number, either as a national number, or in international format starting with a plus sign

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→nextMessageBox() YMessageBox
messagebox.nextMessageBox()

Continues the enumeration of MessageBox interfaces started using `yFirstMessageBox()`.

function **nextMessageBox**( )

Returns :

a pointer to a `YMessageBox` object, corresponding to a MessageBox interface currently online, or a `null` pointer if there are no more MessageBox interfaces to enumerate.

messagebox→registerValueCallback() YMessageBox
messagebox.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.

messagebox→sendFlashMessage() YMessageBox
messagebox.sendFlashMessage()

Sends a Flash SMS (class 0 message).

function **sendFlashMessage**(**recipient**, **message**)

Flash messages are displayed on the handset immediately and are usually not saved on the SIM card. This function can send messages of more than 160 characters, using SMS concatenation. ISO-latin accented characters are supported. For sending messages with special unicode characters such as asian characters and emoticons, use `newMessage` to create a new message and define the content of using methods `addText` et `addUnicodeData`.

Parameters :

recipient a text string with the recipient phone number, either as a national number, or in international format starting with a plus sign

message the text to be sent in the message

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→sendTextMessage() YMessageBox
messagebox.sendTextMessage()

Sends a regular text SMS, with standard parameters.

function **sendTextMessage**(**recipient**, **message**)

This function can send messages of more than 160 characters, using SMS concatenation. ISO-latin accented characters are supported. For sending messages with special unicode characters such as asian characters and emoticons, use `newMessage` to create a new message and define the content of using methods `addText` and `addUnicodeData`.

Parameters :

recipient a text string with the recipient phone number, either as a national number, or in international format starting with a plus sign

message the text to be sent in the message

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→set_logicalName() YMessageBox
messagebox→setLogicalName()
messagebox.set_logicalName()

Changes the logical name of the MessageBox 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 MessageBox interface.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→set_pduReceived() YMessageBox
messagebox→setPduReceived()
messagebox.set_pduReceived()

Changes the value of the incoming SMS units counter.

function **set_pduReceived**(**newval**)

Parameters :

newval an integer corresponding to the value of the incoming SMS units counter

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→set_pduSent() YMessageBox
messagebox→setPduSent()
messagebox.set_pduSent()

Changes the value of the outgoing SMS units counter.

function **set_pduSent**(**newval**)

Parameters :

newval an integer corresponding to the value of the outgoing SMS units counter

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→set_userData() YMessageBox
messagebox→setUserData()
messagebox.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

messagebox→unmuteValueCallbacks() YMessageBox
messagebox.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

messagebox→wait_async() YMessageBox
messagebox.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.40. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');

Global functions

yFindModule(**func**)

Allows you to find a module from its serial number or from its logical name.

yFindModuleInContext(**yctx**, **func**)

Retrieves a module for a given identifier in a YAPI context.

yFirstModule()

Starts the enumeration of modules currently accessible.

YModule methods

module→checkFirmware(**path**, **onlynew**)

Tests whether the byn file is valid for this module.

module→clearCache()

Invalidates the cache.

module→describe()

Returns a descriptive text that identifies the module.

module→download(**pathname**)

Downloads the specified built-in file and returns a binary buffer with its content.

module→functionBaseType(**functionIndex**)

Retrieves the base type of the *n*th function on 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→functionType(**functionIndex**)

Retrieves the type of the *n*th function on the module.

module→functionValue(**functionIndex**)

Retrieves the advertised value of the *n*th function on the module.

module→get_allSettings()

Returns all the settings and uploaded files of the module.

module→get_beacon()
Returns the state of the localization beacon.

module→get_errorMessage()
Returns the error message of the latest error with this module object.

module→get_errorType()
Returns the numerical error code of the latest error with this module object.

module→get_firmwareRelease()
Returns the version of the firmware embedded in the module.

module→get_functionIds(**funType**)
Retrieve all hardware identifier that match the type passed in argument.

module→get_hardwareId()
Returns the unique hardware identifier of the module.

module→get_icon2d()
Returns the icon of the module.

module→get_lastLogs()
Returns a string with last logs of the module.

module→get_logicalName()
Returns the logical name of the module.

module→get_luminosity()
Returns the luminosity of the module informative leds (from 0 to 100).

module→get_parentHub()
Returns the serial number of the YoctoHub on which this module is connected.

module→get_persistentSettings()
Returns the current state of persistent module settings.

module→get_productId()
Returns the USB device identifier of the module.

module→get_productName()
Returns the commercial name of the module, as set by the factory.

module→get_productRelease()
Returns the hardware release version of the module.

module→get_rebootCountdown()
Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()
Returns the serial number of the module, as set by the factory.

module→get_subDevices()
Returns a list of all the modules that are plugged into the current module.

module→get_upTime()
Returns the number of milliseconds spent since the module was powered on.

module→get_url()
Returns the URL used to access the module.

module→get_usbCurrent()
Returns the current consumed by the module on the USB bus, in milli-amps.

module→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

module→get_userVar()
Returns the value previously stored in this attribute.

module→hasFunction(funcId)
Tests if the device includes a specific function.

module→isOnline()
Checks if the module is currently reachable, without raising any error.

module→isOnline_async(callback, context)
Checks if the module is currently reachable, without raising any error.

module→load(msValidity)
Preloads the module cache with a specified validity duration.

module→load_async(msValidity, callback, context)
Preloads the module cache with a specified validity duration (asynchronous version).

module→log(text)
Adds a text message to the device logs.

module→nextModule()
Continues the module enumeration started using `yFirstModule()`.

module→reboot(secBeforeReboot)
Schedules a simple module reboot after the given number of seconds.

module→registerLogCallback(callback)
Registers a device log callback function.

module→revertFromFlash()
Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

module→saveToFlash()
Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(settings)
Restores all the settings of the device.

module→set_allSettingsAndFiles(settings)
Restores all the settings and uploaded files to the module.

module→set_beacon(newval)
Turns on or off the module localization beacon.

module→set_logicalName(newval)
Changes the logical name of the module.

module→set_luminosity(newval)
Changes the luminosity of the module informative leds.

module→set_userData(data)
Stores a user context provided as argument in the userData attribute of the function.

module→set_userVar(newval)
Stores a 32 bit value in the device RAM.

module→triggerFirmwareUpdate(secBeforeReboot)
Schedules a module reboot into special firmware update mode.

module→updateFirmware(path)
Prepares a firmware update of the module.

module→updateFirmwareEx(path, force)
Prepares a firmware update of the module.

module→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YModule.FindModule() YModule yFindModule()yFindModule()

Allows you to find a module from its serial number or from its logical name.

function **FindModule**(**func**)

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string containing either the serial number or the logical name of the desired module

Returns :

a `YModule` object allowing you to drive the module or get additional information on the module.

YModule.FindModuleInContext() YModule
yFindModuleInContext()

Retrieves a module for a given identifier in a YAPI context.

function **FindModuleInContext**(**yctx**, **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 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 :

yctx a YAPI context

func a string that uniquely characterizes the module

Returns :

a `YModule` object allowing you to drive the module.

YModule.FirstModule() YModule yFirstModule()yFirstModule()

Starts the enumeration of modules currently accessible.

function **FirstModule**( )

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

module→checkFirmware()module.checkFirmware() YModule

Tests whether the byn file is valid for this module.

function **checkFirmware**(**path**, **onlynew**)

This method is useful to test if the module needs to be updated. It is possible to pass a directory as argument instead of a file. In this case, this method returns the path of the most recent appropriate `.byn` file. If the parameter `onlynew` is true, the function discards firmwares that are older or equal to the installed firmware.

Parameters :

path	the path of a byn file or a directory that contains byn files
onlynew	returns only files that are strictly newer

Returns :

the path of the byn file to use or a empty string if no byn files matches the requirement

On failure, throws an exception or returns a string that start with "error:".

module→clearCache()module.clearCache() YModule

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the module attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

module→describe()module.describe() YModule

Returns a descriptive text that identifies the module.

```
function describe( )
```

The text may include either the logical name or the serial number of the module.

Returns :
a string that describes the module

module→download()module.download() YModule

Downloads the specified built-in file and returns a binary buffer with its content.

function **download**(**pathname**)

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→functionBaseType() YModule
module.functionBaseType()

Retrieves the base type of the *n*th function on the module.

```
function functionBaseType( functionIndex)
```

For instance, the base type of all measuring functions is "Sensor".

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 base type of the function

On failure, throws an exception or returns an empty string.

module→functionCount()module.functionCount() YModule

Returns the number of functions (beside the "module" interface) available on the module.

function **functionCount**( )

Returns :
the number of functions on the module

On failure, throws an exception or returns a negative error code.

module→functionId()module.functionId() YModule

Retrieves the hardware identifier of the *n*th function on the module.

function **functionId**(**functionIndex**)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

module→functionName()module.functionName() YModule

Retrieves the logical name of the *n*th function on the module.

function **functionName**(**functionIndex**)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

module→functionType()module.functionType() YModule

Retrieves the type of the *n*th function on the module.

function **functionType**(**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 type of the function

On failure, throws an exception or returns an empty string.

module→functionValue()module.functionValue() YModule

Retrieves the advertised value of the *n*th function on the module.

function **functionValue**(**functionIndex**)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

module→get_allSettings() YModule
module→allSettings()module.get_allSettings()

Returns all the settings and uploaded files of the module.

function **get_allSettings**( )

Useful to backup all the logical names, calibrations parameters, and uploaded files of a device.

Returns :
a binary buffer with all the settings.

On failure, throws an exception or returns an binary object of size 0.

module→get_beacon() YModule
module→beacon()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() YModule
module→errorMessage()module.get_errorMessage()

Returns the error message of the latest error with this module object.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using this module object

module→get_errorType() YModule
module→errorType()module.get_errorType()

Returns the numerical error code of the latest error with this module object.

```
function get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using this module object

module→get_firmwareRelease() YModule
module→firmwareRelease()
module.get_firmwareRelease()

Returns the version of the firmware embedded in the module.

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_functionIds() YModule
module→functionIds()module.get_functionIds()

Retrieve all hardware identifier that match the type passed in argument.

function **get_functionIds**(**funType**)

Parameters :

funType The type of function (Relay, LightSensor, Voltage,...)

Returns :

an array of strings.

module→get_hardwareId() YModule
module→hardwareId()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

module→get_icon2d() YModule
module→icon2d()module.get_icon2d()

Returns the icon of the module.

function **get_icon2d**( )

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format. On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_lastLogs() YModule
module→lastLogs()module.get_lastLogs()

Returns a string with last logs of the module.

function **get_lastLogs**( )

This method return only logs that are still in the module.

Returns :
a string with last logs of the module. On failure, throws an exception or returns YAPI_INVALID_STRING.

module→get_logicalName() YModule
module→logicalName()module.get_logicalName()

Returns the logical name of the module.

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() YModule
module→luminosity()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() YModule
module→persistentSettings()
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() YModule
module→productId()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() YModule
module→productName()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() YModule
module→productRelease()
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() YModule
module→rebootCountdown()
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() YModule
module→serialNumber()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() YModule
module→upTime()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_usbCurrent() YModule
module→usbCurrent()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() YModule
module→userData()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→get_userVar() YModule
module→userVar()module.get_userVar()

Returns the value previously stored in this attribute.

function **get_userVar**( )

On startup and after a device reboot, the value is always reset to zero.

Returns :
an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns Y_USERVAR_INVALID.

module→hasFunction()module.hasFunction() YModule

Tests if the device includes a specific function.

```
function hasFunction( funcId)
```

This method takes a function identifier and returns a boolean.

Parameters :

funcId the requested function identifier

Returns :

true if the device has the function identifier

module→isOnline()module.isOnline() YModule

Checks if the module is currently reachable, without raising any error.

function **isOnline**( )

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

`true` if the module can be reached, and `false` otherwise

module→load()module.load() YModule

Preloads the module cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→log()module.log() YModule

Adds a text message to the device logs.

```
function log( text)
```

This function is useful in particular to trace the execution of HTTP callbacks. If a newline is desired after the message, it must be included in the string.

Parameters :

text the string to append to the logs.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→nextModule()module.nextModule() YModule

Continues the module enumeration started using `yFirstModule()`.

function **nextModule**( )

Returns :

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot()module.reboot() YModule

Schedules a simple module reboot after the given number of seconds.

function **reboot**(**secBeforeReboot**)

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→revertFromFlash() YModule
module.revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

function **revertFromFlash**( )

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→saveToFlash()module.saveToFlash() YModule

Saves current settings in the nonvolatile memory of the module.

function **saveToFlash**( )

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettings() YModule
module→setAllSettings()module.set_allSettings()

Restores all the settings of the device.

function **set_allSettings**(**settings**)

Useful to restore all the logical names and calibrations parameters of a module from a backup.Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettingsAndFiles() YModule
module→setAllSettingsAndFiles()
module.set_allSettingsAndFiles()

Restores all the settings and uploaded files to the module.

function **set_allSettingsAndFiles**(**settings**)

This method is useful to restore all the logical names and calibrations parameters, uploaded files etc. of a device from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_beacon() YModule
module→setBeacon()module.set_beacon()

Turns on or off the module localization beacon.

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() YModule
module→setLogicalName()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() YModule
module→setLuminosity()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_userData() YModule
module→setUserData()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→set_userVar() YModule
module→setUserVar()module.set_userVar()

Stores a 32 bit value in the device RAM.

function **set_userVar**(**newval**)

This attribute is at programmer disposal, should he need to store a state variable. On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→triggerFirmwareUpdate() YModule
module.triggerFirmwareUpdate()

Schedules a module reboot into special firmware update mode.

function **triggerFirmwareUpdate**(**secBeforeReboot**)

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→updateFirmware()module.updateFirmware() YModule

Prepares a firmware update of the module.

function **updateFirmware**(**path**)

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

Parameters :

path the path of the `.byn` file to use.

Returns :

a `YFirmwareUpdate` object or NULL on error.

module→updateFirmwareEx() YModule
module.updateFirmwareEx()

Prepares a firmware update of the module.

function **updateFirmwareEx**(**path**, **force**)

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

Parameters :

path the path of the `.byn` file to use.

force true to force the firmware update even if some prerequisites appear not to be met

Returns :

a `YFirmwareUpdate` object or NULL on error.

module→wait_async()module.wait_async() YModule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.41. Motor function interface

Yoctopuce application programming interface allows you to drive the power sent to the motor to make it turn both ways, but also to drive accelerations and decelerations. The motor will then accelerate automatically: you will not have to monitor it. The API also allows to slow down the motor by shortening its terminals: the motor will then act as an electromagnetic brake.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_motor.js'></script>
cpp     #include "yocto_motor.h"
m       #import "yocto_motor.h"
pas     uses yocto_motor;
vb      yocto_motor.vb
cs      yocto_motor.cs
java    import com.yoctopuce.YoctoAPI.YMotor;
uwp     import com.yoctopuce.YoctoAPI.YMotor;
py      from yocto_motor import *
php     require_once('yocto_motor.php');
es      in HTML: <script src="../../lib/yocto_motor.js"></script>
        in node.js: require('yoctolib-es2017/yocto_motor.js');
```

Global functions
yFindMotor(**func**) Retrieves a motor for a given identifier.
yFindMotorInContext(**yctx**, **func**) Retrieves a motor for a given identifier in a YAPI context.
yFirstMotor() Starts the enumeration of motors currently accessible.
yFirstMotorInContext(**yctx**) Starts the enumeration of motors currently accessible.
`YMotor` methods
motor→brakingForceMove(**targetPower**, **delay**) Changes progressively the braking force applied to the motor for a specific duration.
motor→clearCache() Invalidates the cache.
motor→describe() Returns a short text that describes unambiguously the instance of the motor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
motor→drivingForceMove(**targetPower**, **delay**) Changes progressively the power sent to the moteur for a specific duration.
motor→get_advertisedValue() Returns the current value of the motor (no more than 6 characters).
motor→get_brakingForce() Returns the braking force applied to the motor, as a percentage.
motor→get_cutOffVoltage() Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.
motor→get_drivingForce() Returns the power sent to the motor, as a percentage between -100% and +100%.

motor→get_errorMessage()

Returns the error message of the latest error with the motor.

motor→get_errorType()

Returns the numerical error code of the latest error with the motor.

motor→get_failSafeTimeout()

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

motor→get_frequency()

Returns the PWM frequency used to control the motor.

motor→get_friendlyName()

Returns a global identifier of the motor in the format `MODULE_NAME.FUNCTION_NAME`.

motor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

motor→get_functionId()

Returns the hardware identifier of the motor, without reference to the module.

motor→get_hardwareId()

Returns the unique hardware identifier of the motor in the form `SERIAL.FUNCTIONID`.

motor→get_logicalName()

Returns the logical name of the motor.

motor→get_module()

Gets the `YModule` object for the device on which the function is located.

motor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

motor→get_motorStatus()

Return the controller state.

motor→get_overCurrentLimit()

Returns the current threshold (in mA) above which the controller automatically switches to error state.

motor→get_starterTime()

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

motor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

motor→isOnline()

Checks if the motor is currently reachable, without raising any error.

motor→isOnline_async(**callback**, **context**)

Checks if the motor is currently reachable, without raising any error (asynchronous version).

motor→keepALive()

Rearms the controller failsafe timer.

motor→load(**msValidity**)

Preloads the motor cache with a specified validity duration.

motor→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

motor→load_async(**msValidity**, **callback**, **context**)

Preloads the motor cache with a specified validity duration (asynchronous version).

motor→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

motor→nextMotor()

Continues the enumeration of motors started using `yFirstMotor()`.

motor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

motor→resetStatus()

Reset the controller state to IDLE.

motor→set_brakingForce(newval)

Changes immediately the braking force applied to the motor (in percents).

motor→set_cutOffVoltage(newval)

Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

motor→set_drivingForce(newval)

Changes immediately the power sent to the motor.

motor→set_failSafeTimeout(newval)

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

motor→set_frequency(newval)

Changes the PWM frequency used to control the motor.

motor→set_logicalName(newval)

Changes the logical name of the motor.

motor→set_overCurrentLimit(newval)

Changes the current threshold (in mA) above which the controller automatically switches to error state.

motor→set_starterTime(newval)

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

motor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

motor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

motor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMotor.FindMotor() YMotor yFindMotor()yFindMotor()

Retrieves a motor for a given identifier.

```
function FindMotor( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the motor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMotor.isOnline()` to test if the motor is indeed online at a given time. In case of ambiguity when looking for a motor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the motor

Returns :

a `YMotor` object allowing you to drive the motor.

YMotor.FindMotorInContext() YMotor yFindMotorInContext()

Retrieves a motor for a given identifier in a YAPI context.

function **FindMotorInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the motor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMotor.isOnline()` to test if the motor is indeed online at a given time. In case of ambiguity when looking for a motor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the motor

Returns :

a `YMotor` object allowing you to drive the motor.

YMotor.FirstMotor() YMotor yFirstMotor()yFirstMotor()

Starts the enumeration of motors currently accessible.

function **FirstMotor**( )

Use the method `YMotor.nextMotor()` to iterate on next motors.

Returns :

a pointer to a `YMotor` object, corresponding to the first motor currently online, or a `null` pointer if there are none.

YMotor.FirstMotorInContext() YMotor yFirstMotorInContext()

Starts the enumeration of motors currently accessible.

function **FirstMotorInContext**(**yctx**)

Use the method `YMotor.nextMotor()` to iterate on next motors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YMotor` object, corresponding to the first motor currently online, or a `null` pointer if there are none.

motor→brakingForceMove() YMotor
motor.brakingForceMove()

Changes progressively the braking force applied to the motor for a specific duration.

function **brakingForceMove**(**targetPower**, **delay**)

Parameters :

targetPower	desired braking force, in percents
delay	duration (in ms) of the transition

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→clearCache()motor.clearCache() YMotor

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the motor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

motor→describe()motor.describe() YMotor

Returns a short text that describes unambiguously the instance of the motor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the motor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

motor→drivingForceMove() YMotor
motor.drivingForceMove()

Changes progressively the power sent to the moteur for a specific duration.

function **drivingForceMove**(**targetPower**, **delay**)

Parameters :

targetPower	desired motor power, in percents (between -100% and +100%)
delay	duration (in ms) of the transition

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→get_advertisedValue() YMotor
motor→advertisedValue()
motor.get_advertisedValue()

Returns the current value of the motor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the motor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

motor→get_brakingForce() YMotor
motor→brakingForce()motor.get_brakingForce()

Returns the braking force applied to the motor, as a percentage.

function **get_brakingForce**( )

The value 0 corresponds to no braking (free wheel).

Returns :
a floating point number corresponding to the braking force applied to the motor, as a percentage

On failure, throws an exception or returns `Y_BRAKINGFORCE_INVALID`.

motor→get_cutOffVoltage() YMotor
motor→cutOffVoltage()motor.get_cutOffVoltage()

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

function **get_cutOffVoltage**( )

This setting prevents damage to a battery that can occur when drawing current from an "empty" battery.

Returns :

a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

On failure, throws an exception or returns `Y_CUTOFFVOLTAGE_INVALID`.

motor→get_drivingForce() YMotor
motor→drivingForce()motor.get_drivingForce()

Returns the power sent to the motor, as a percentage between -100% and +100%.

function **get_drivingForce**( )

Returns :

a floating point number corresponding to the power sent to the motor, as a percentage between -100% and +100%

On failure, throws an exception or returns `Y_DRIVINGFORCE_INVALID`.

motor→get_errorMessage() YMotor
motor→errorMessage()motor.get_errorMessage()

Returns the error message of the latest error with the motor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the motor object

motor→get_errorType() YMotor
motor→errorType()motor.get_errorType()

Returns the numerical error code of the latest error with the motor.

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 the motor object

motor→get_failSafeTimeout() YMotor
motor→failSafeTimeout()motor.get_failSafeTimeout()

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

function **get_failSafeTimeout**( )

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

Returns :

an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

On failure, throws an exception or returns `Y_FAILSAFETIMEOUT_INVALID`.

motor→get_frequency() YMotor
motor→frequency()motor.get_frequency()

Returns the PWM frequency used to control the motor.

function **get_frequency**( )

Returns :
a floating point number corresponding to the PWM frequency used to control the motor

On failure, throws an exception or returns Y_FREQUENCY_INVALID.

motor→get_friendlyName() YMotor
motor→friendlyName()motor.get_friendlyName()

Returns a global identifier of the motor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the motor if they are defined, otherwise the serial number of the module and the hardware identifier of the motor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the motor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

motor→get_functionDescriptor() YMotor
motor→functionDescriptor()
motor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

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.

motor→get_functionId() YMotor
motor→functionId()motor.get_functionId()

Returns the hardware identifier of the motor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the motor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

motor→get_hardwareId() YMotor
motor→hardwareId()motor.get_hardwareId()

Returns the unique hardware identifier of the motor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the motor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the motor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

motor→get_logicalName() YMotor
motor→logicalName()motor.get_logicalName()

Returns the logical name of the motor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the motor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

motor→get_module() YMotor
motor→module()motor.get_module()

Gets the YModule object for the device on which the function is located.

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

motor→get_motorStatus() YMotor
motor→motorStatus()motor.get_motorStatus()

Return the controller state.

function **get_motorStatus**( )

Possible states are: IDLE when the motor is stopped/in free wheel, ready to start; FORWD when the controller is driving the motor forward; BACKWD when the controller is driving the motor backward; BRAKE when the controller is braking; LOVOLT when the controller has detected a low voltage condition; HICURR when the controller has detected an overcurrent condition; HIHEAT when the controller has detected an overheat condition; FAILSF when the controller switched on the failsafe security.

When an error condition occurred (LOVOLT, HICURR, HIHEAT, FAILSF), the controller status must be explicitly reset using the `resetStatus` function.

Returns :

a value among `Y_MOTORSTATUS_IDLE`, `Y_MOTORSTATUS_BRAKE`, `Y_MOTORSTATUS_FORWD`, `Y_MOTORSTATUS_BACKWD`, `Y_MOTORSTATUS_LOVOLT`, `Y_MOTORSTATUS_HICURR`, `Y_MOTORSTATUS_HIHEAT` and `Y_MOTORSTATUS_FAILSF`

On failure, throws an exception or returns `Y_MOTORSTATUS_INVALID`.

motor→get_overCurrentLimit() YMotor
motor→overCurrentLimit()
motor.get_overCurrentLimit()

Returns the current threshold (in mA) above which the controller automatically switches to error state.

function **get_overCurrentLimit**( )

A zero value means that there is no limit.

Returns :

an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

On failure, throws an exception or returns `Y_OVERCURRENTLIMIT_INVALID`.

motor→get_starterTime() YMotor
motor→starterTime()motor.get_starterTime()

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

function **get_starterTime**( )

Returns :

an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

On failure, throws an exception or returns `Y_STARTERTIME_INVALID`.

motor→get_userData() YMotor
motor→userData()motor.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

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.

motor→isOnline()motor.isOnline() YMotor

Checks if the motor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the motor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the motor.

Returns :

`true` if the motor can be reached, and `false` otherwise

motor→keepALive()motor.keepALive() YMotor

Rearms the controller failsafe timer.

function **keepALive**( )

When the motor is running and the failsafe feature is active, this function should be called periodically to prove that the control process is running properly. Otherwise, the motor is automatically stopped after the specified timeout. Calling a motor *set* function implicitely rearms the failsafe timer.

motor→load()motor.load() YMotor

Preloads the motor 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.

motor→loadAttribute()motor.loadAttribute() YMotor

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

motor→muteValueCallbacks() YMotor
motor.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→nextMotor()motor.nextMotor() YMotor

Continues the enumeration of motors started using `yFirstMotor()`.

function **nextMotor**( )

Returns :

a pointer to a `YMotor` object, corresponding to a motor currently online, or a `null` pointer if there are no more motors to enumerate.

motor→registerValueCallback() YMotor
motor.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.

motor→resetStatus()motor.resetStatus() YMotor

Reset the controller state to IDLE.

function **resetStatus**( )

This function must be invoked explicitely after any error condition is signaled.

motor→set_brakingForce() YMotor
motor→setBrakingForce()motor.set_brakingForce()

Changes immediately the braking force applied to the motor (in percents).

function **set_brakingForce**(**newval**)

The value 0 corresponds to no braking (free wheel). When the braking force is changed, the driving power is set to zero. The value is a percentage.

Parameters :

newval a floating point number corresponding to immediately the braking force applied to the motor (in percents)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_cutOffVoltage() YMotor
motor→setCutOffVoltage()motor.set_cutOffVoltage()

Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

function **set_cutOffVoltage**(**newval**)

This setting prevent damage to a battery that can occur when drawing current from an "empty" battery. Note that whatever the cutoff threshold, the controller switches to undervoltage error state if the power supply goes under 3V, even for a very brief time.

Parameters :

newval a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_drivingForce() YMotor
motor→setDrivingForce()motor.set_drivingForce()

Changes immediately the power sent to the motor.

function **set_drivingForce**(**newval**)

The value is a percentage between -100% to 100%. If you want go easy on your mechanics and avoid excessive current consumption, try to avoid brutal power changes. For example, immediate transition from forward full power to reverse full power is a very bad idea. Each time the driving power is modified, the braking power is set to zero.

Parameters :

newval a floating point number corresponding to immediately the power sent to the motor

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_failSafeTimeout() YMotor
motor→setFailSafeTimeout()
motor.set_failSafeTimeout()

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

function **set_failSafeTimeout**(**newval**)

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

Parameters :

newval an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_frequency() YMotor
motor→setFrequency()motor.set_frequency()

Changes the PWM frequency used to control the motor.

```
function set_frequency( newval)
```

Low frequency is usually more efficient and may help the motor to start, but an audible noise might be generated. A higher frequency reduces the noise, but more energy is converted into heat.

Parameters :

newval a floating point number corresponding to the PWM frequency used to control the motor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_logicalName() YMotor
motor→setLogicalName()motor.set_logicalName()

Changes the logical name of the motor.

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 motor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_overCurrentLimit() YMotor
motor→setOverCurrentLimit()
motor.set_overCurrentLimit()

Changes the current threshold (in mA) above which the controller automatically switches to error state.

function **set_overCurrentLimit**(**newval**)

A zero value means that there is no limit. Note that whatever the current limit is, the controller switches to OVERCURRENT status if the current goes above 32A, even for a very brief time.

Parameters :

newval an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_starterTime() YMotor
motor→setStarterTime()motor.set_starterTime()

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

function **set_starterTime**(**newval**)

Parameters :

newval an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_userData() YMotor
motor→setUserData()motor.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

```
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

motor→unmuteValueCallbacks() YMotor
motor.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→wait_async()motor.wait_async() YMotor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.42. MultiAxisController function interface

The Yoctopuce application programming interface allows you to drive a stepper motor.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_multiaxiscontroller.js'></script>
cpp	#include "yocto_multiaxiscontroller.h"
m	#import "yocto_multiaxiscontroller.h"
pas	uses yocto_multiaxiscontroller;
vb	yocto_multiaxiscontroller.vb
cs	yocto_multiaxiscontroller.cs
java	import com.yoctopuce.YoctoAPI.YMultiAxisController;
uwp	import com.yoctopuce.YoctoAPI.YMultiAxisController;
py	from yocto_multiaxiscontroller import *
php	require_once('yocto_multiaxiscontroller.php');
es	in HTML: <script src="../../lib/yocto_multiaxiscontroller.js"></script> in node.js: require('yoctolib-es2017/yocto_multiaxiscontroller.js');

Global functions

yFindMultiAxisController(**func**)

Retrieves a multi-axis controller for a given identifier.

yFindMultiAxisControllerInContext(**yctx**, **func**)

Retrieves a multi-axis controller for a given identifier in a YAPI context.

yFirstMultiAxisController()

Starts the enumeration of multi-axis controllers currently accessible.

yFirstMultiAxisControllerInContext(**yctx**)

Starts the enumeration of multi-axis controllers currently accessible.

`YMultiAxisController` methods

multiaxiscontroller→abortAndBrake()

Stops the motor smoothly as soon as possible, without waiting for ongoing move completion.

multiaxiscontroller→abortAndHiZ()

Turn the controller into Hi-Z mode immediately, without waiting for ongoing move completion.

multiaxiscontroller→clearCache()

Invalidates the cache.

multiaxiscontroller→describe()

Returns a short text that describes unambiguously the instance of the multi-axis controller in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

multiaxiscontroller→emergencyStop()

Stops the motor with an emergency alert, without taking any additional precaution.

multiaxiscontroller→findHomePosition(**speed**)

Starts all motors backward at the specified speeds, to search for the motor home position.

multiaxiscontroller→get_advertisedValue()

Returns the current value of the multi-axis controller (no more than 6 characters).

multiaxiscontroller→get_errorMessage()

Returns the error message of the latest error with the multi-axis controller.

multiaxiscontroller→get_errorType()

Returns the numerical error code of the latest error with the multi-axis controller.

multiaxiscontroller→get_friendlyName()

Returns a global identifier of the multi-axis controller in the format MODULE_NAME.FUNCTION_NAME.

multiaxiscontroller→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

multiaxiscontroller→get_functionId()

Returns the hardware identifier of the multi-axis controller, without reference to the module.

multiaxiscontroller→get_globalState()

Returns the stepper motor set overall state.

multiaxiscontroller→get_hardwareId()

Returns the unique hardware identifier of the multi-axis controller in the form SERIAL.FUNCTIONID.

multiaxiscontroller→get_logicalName()

Returns the logical name of the multi-axis controller.

multiaxiscontroller→get_module()

Gets the YModule object for the device on which the function is located.

multiaxiscontroller→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

multiaxiscontroller→get_nAxis()

Returns the number of synchronized controllers.

multiaxiscontroller→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

multiaxiscontroller→isOnline()

Checks if the multi-axis controller is currently reachable, without raising any error.

multiaxiscontroller→isOnline_async(callback, context)

Checks if the multi-axis controller is currently reachable, without raising any error (asynchronous version).

multiaxiscontroller→load(msValidity)

Preloads the multi-axis controller cache with a specified validity duration.

multiaxiscontroller→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

multiaxiscontroller→load_async(msValidity, callback, context)

Preloads the multi-axis controller cache with a specified validity duration (asynchronous version).

multiaxiscontroller→moveRel(relPos)

Starts all motors synchronously to reach a given relative position.

multiaxiscontroller→moveTo(absPos)

Starts all motors synchronously to reach a given absolute position.

multiaxiscontroller→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

multiaxiscontroller→nextMultiAxisController()

Continues the enumeration of multi-axis controllers started using yFirstMultiAxisController().

multiaxiscontroller→pause(waitMs)

Keep the motor in the same state for the specified amount of time, before processing next command.

multiaxiscontroller→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

multiaxiscontroller→reset()

Reinitialize all controllers and clear all alert flags.

multiaxiscontroller→set_logicalName(newval)

Changes the logical name of the multi-axis controller.

multiaxiscontroller→set_nAxis(newval)

Changes the number of synchronized controllers.

multiaxiscontroller→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

multiaxiscontroller→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

multiaxiscontroller→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMultiAxisController.FindMultiAxisController() YMultiAxisController yFindMultiAxisController()yFindMultiAxisController()

Retrieves a multi-axis controller for a given identifier.

```
function FindMultiAxisController( 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 multi-axis controller is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMultiAxisController.isOnline()` to test if the multi-axis controller is indeed online at a given time. In case of ambiguity when looking for a multi-axis controller 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the multi-axis controller

Returns :

a `YMultiAxisController` object allowing you to drive the multi-axis controller.

YMultiAxisController.FindMultiAxisControllerInContext() YMultiAxisController yFindMultiAxisControllerInContext()

Retrieves a multi-axis controller for a given identifier in a YAPI context.

function **FindMultiAxisControllerInContext**(**yctx**, **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 multi-axis controller is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMultiAxisController.isOnline()` to test if the multi-axis controller is indeed online at a given time. In case of ambiguity when looking for a multi-axis controller 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 :

yctx a YAPI context

func a string that uniquely characterizes the multi-axis controller

Returns :

a `YMultiAxisController` object allowing you to drive the multi-axis controller.

YMultiAxisController.FirstMultiAxisController() YMultiAxisController yFirstMultiAxisController()yFirstMultiAxisController()

Starts the enumeration of multi-axis controllers currently accessible.

function **FirstMultiAxisController**( )

Use the method `YMultiAxisController.nextMultiAxisController()` to iterate on next multi-axis controllers.

Returns :

a pointer to a `YMultiAxisController` object, corresponding to the first multi-axis controller currently online, or a `null` pointer if there are none.

YMultiAxisController.FirstMultiAxisControllerInContext() YMultiAxisController yFirstMultiAxisControllerInContext()

Starts the enumeration of multi-axis controllers currently accessible.

function **FirstMultiAxisControllerInContext**(**yctx**)

Use the method `YMultiAxisController.nextMultiAxisController()` to iterate on next multi-axis controllers.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YMultiAxisController` object, corresponding to the first multi-axis controller currently online, or a `null` pointer if there are none.

multiaxiscontroller→abortAndBrake() YMultiAxisController
multiaxiscontroller.abortAndBrake()

Stops the motor smoothly as soon as possible, without waiting for ongoing move completion.

function **abortAndBrake**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→abortAndHiZ() YMultiAxisController
multiaxiscontroller.abortAndHiZ()

Turn the controller into Hi-Z mode immediately, without waiting for ongoing move completion.

function **abortAndHiZ**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→clearCache() YMultiAxisController
multiaxiscontroller.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the multi-axis controller attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

multiaxiscontroller→describe() YMultiAxisController
multiaxiscontroller.describe()

Returns a short text that describes unambiguously the instance of the multi-axis controller in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the multi-axis controller (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

multiaxiscontroller→emergencyStop() YMultiAxisController
multiaxiscontroller.emergencyStop()

Stops the motor with an emergency alert, without taking any additional precaution.

function **emergencyStop**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→findHomePosition() YMultiAxisController
multiaxiscontroller.findHomePosition()

Starts all motors backward at the specified speeds, to search for the motor home position.

function **findHomePosition**(**speed**)

Parameters :

speed desired speed for all axis, in steps per second.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→get_advertisedValue() YMultiAxisController
multiaxiscontroller→advertisedValue()
multiaxiscontroller.get_advertisedValue()

Returns the current value of the multi-axis controller (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the multi-axis controller (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

multiaxiscontroller→get_errorMessage() YMultiAxisController
multiaxiscontroller→errorMessage()
multiaxiscontroller.get_errorMessage()

Returns the error message of the latest error with the multi-axis controller.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the multi-axis controller object

multiaxiscontroller→get_errorType() YMultiAxisController
multiaxiscontroller→errorType()
multiaxiscontroller.get_errorType()

Returns the numerical error code of the latest error with the multi-axis controller.

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 the multi-axis controller object

multiaxiscontroller→get_friendlyName() YMultiAxisController
multiaxiscontroller→friendlyName()
multiaxiscontroller.get_friendlyName()

Returns a global identifier of the multi-axis controller in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the multi-axis controller if they are defined, otherwise the serial number of the module and the hardware identifier of the multi-axis controller (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the multi-axis controller using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

multiaxiscontroller→get_functionDescriptor() YMultiAxisController
multiaxiscontroller→functionDescriptor()
multiaxiscontroller.get_functionDescriptor()

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.

multiaxiscontroller→get_functionId() YMultiAxisController
multiaxiscontroller→functionId()
multiaxiscontroller.get_functionId()

Returns the hardware identifier of the multi-axis controller, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the multi-axis controller (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

multiaxiscontroller→get_globalState() YMultiAxisController
multiaxiscontroller→globalState()
multiaxiscontroller.get_globalState()

Returns the stepper motor set overall state.

function **get_globalState**( )

Returns :

a value among Y_GLOBALSTATE_ABSENT, Y_GLOBALSTATE_ALERT, Y_GLOBALSTATE_HI_Z, Y_GLOBALSTATE_STOP, Y_GLOBALSTATE_RUN and Y_GLOBALSTATE_BATCH corresponding to the stepper motor set overall state

On failure, throws an exception or returns Y_GLOBALSTATE_INVALID.

multiaxiscontroller→get_hardwareId() YMultiAxisController
multiaxiscontroller→hardwareId()
multiaxiscontroller.get_hardwareId()

Returns the unique hardware identifier of the multi-axis controller in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the multi-axis controller (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the multi-axis controller (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

multiaxiscontroller→get_logicalName() YMultiAxisController
multiaxiscontroller→logicalName()
multiaxiscontroller.get_logicalName()

Returns the logical name of the multi-axis controller.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the multi-axis controller.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

multiaxiscontroller→get_module() YMultiAxisController
multiaxiscontroller→module()
multiaxiscontroller.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

multiaxiscontroller→get_nAxis() YMultiAxisController
multiaxiscontroller→nAxis()
multiaxiscontroller.get_nAxis()

Returns the number of synchronized controllers.

function **get_nAxis**( )

Returns :
an integer corresponding to the number of synchronized controllers

On failure, throws an exception or returns `Y_NAXIS_INVALID`.

multiaxiscontroller→get_userData() YMultiAxisController
multiaxiscontroller→userData()
multiaxiscontroller.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.

multiaxiscontroller→isOnline() YMultiAxisController
multiaxiscontroller.isOnline()

Checks if the multi-axis controller is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the multi-axis controller 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 multi-axis controller.

Returns :

`true` if the multi-axis controller can be reached, and `false` otherwise

multiaxiscontroller→load()multiaxiscontroller.load() YMultiAxisController

Preloads the multi-axis controller 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.

multiaxiscontroller→loadAttribute() YMultiAxisController
multiaxiscontroller.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

multiaxiscontroller→moveRel() YMultiAxisController
multiaxiscontroller.moveRel()

Starts all motors synchronously to reach a given relative position.

function **moveRel**(**relPos**)

The time needed to reach the requested position will depend on the lowest acceleration and max speed parameters configured for all motors. The final position will be reached on all axis at the same time.

Parameters :

relPos relative position, measured in steps from the current position.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→moveTo() YMultiAxisController
multiaxiscontroller.moveTo()

Starts all motors synchronously to reach a given absolute position.

function **moveTo**(**absPos**)

The time needed to reach the requested position will depend on the lowest acceleration and max speed parameters configured for all motors. The final position will be reached on all axis at the same time.

Parameters :

absPos absolute position, measured in steps from each origin.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→muteValueCallbacks() YMultiAxisController
multiaxiscontroller.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

multiaxiscontroller→nextMultiAxisController() YMultiAxisController
multiaxiscontroller.nextMultiAxisController()

Continues the enumeration of multi-axis controllers started using `yFirstMultiAxisController()`.

function **nextMultiAxisController**( )

Returns :

a pointer to a `YMultiAxisController` object, corresponding to a multi-axis controller currently online, or a `null` pointer if there are no more multi-axis controllers to enumerate.

multiaxiscontroller→pause() YMultiAxisController
multiaxiscontroller.pause()

Keep the motor in the same state for the specified amount of time, before processing next command.

function **pause**(**waitMs**)

Parameters :

waitMs wait time, specified in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→registerValueCallback() YMultiAxisController
multiaxiscontroller.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.

multiaxiscontroller→reset()multiaxiscontroller.reset() YMultiAxisController

Reinitialize all controllers and clear all alert flags.

function **reset**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

multiaxiscontroller→set_logicalName() YMultiAxisController
multiaxiscontroller→setLogicalName()
multiaxiscontroller.set_logicalName()

Changes the logical name of the multi-axis controller.

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 multi-axis controller.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

multiaxiscontroller→set_nAxis() YMultiAxisController
multiaxiscontroller→setNAxis()
multiaxiscontroller.set_nAxis()

Changes the number of synchronized controllers.

function **set_nAxis**(**newval**)

Parameters :

newval an integer corresponding to the number of synchronized controllers

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

multiaxiscontroller→set_userData() YMultiAxisController
multiaxiscontroller→setUserData()
multiaxiscontroller.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

multiaxiscontroller→unmuteValueCallbacks() YMultiAxisController
multiaxiscontroller.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

multiaxiscontroller→wait_async() YMultiAxisController
multiaxiscontroller.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.43. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_network.js'></script>
cpp	#include "yocto_network.h"
m	#import "yocto_network.h"
pas	uses yocto_network;
vb	yocto_network.vb
cs	yocto_network.cs
java	import com.yoctopuce.YoctoAPI.YNetwork;
uwp	import com.yoctopuce.YoctoAPI.YNetwork;
py	from yocto_network import *
php	require_once('yocto_network.php');
es	in HTML: <script src="../../lib/yocto_network.js"></script> in node.js: require('yoctolib-es2017/yocto_network.js');

Global functions

yFindNetwork(**func**)

Retrieves a network interface for a given identifier.

yFindNetworkInContext(**yctx**, **func**)

Retrieves a network interface for a given identifier in a YAPI context.

yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

yFirstNetworkInContext(**yctx**)

Starts the enumeration of network interfaces currently accessible.

`YNetwork` methods

network→callbackLogin(**username**, **password**)

Connects to the notification callback and saves the credentials required to log into it.

network→clearCache()

Invalidates the cache.

network→describe()

Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

network→get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

network→get_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

network→get_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

network→get_callbackEncoding()

Returns the encoding standard to use for representing notification values.

network→get_callbackInitialDelay()

Returns the initial waiting time before first callback notifications, in seconds.

network→get_callbackMaxDelay()

Returns the waiting time between two HTTP callbacks when there is nothing new.

network→get_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

network→get_callbackMinDelay()

Returns the minimum waiting time between two HTTP callbacks, in seconds.

network→get_callbackSchedule()

Returns the HTTP callback schedule strategy, as a text string.

network→get_callbackUrl()

Returns the callback URL to notify of significant state changes.

network→get_defaultPage()

Returns the HTML page to serve for the URL "/"" of the hub.

network→get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→get_errorMessage()

Returns the error message of the latest error with the network interface.

network→get_errorType()

Returns the numerical error code of the latest error with the network interface.

network→get_friendlyName()

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

network→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

network→get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

network→get_hardwareId()

Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

network→get_httpPort()

Returns the HTML page to serve for the URL "/"" of the hub.

network→get_ipAddress()

Returns the IP address currently in use by the device.

network→get_ipConfig()

Returns the IP configuration of the network interface.

network→get_logicalName()

Returns the logical name of the network interface.

network→get_macAddress()

Returns the MAC address of the network interface.

network→get_module()

Gets the `YModule` object for the device on which the function is located.

network→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

network→get_ntpServer()

Returns the IP address of the NTP server to be used by the device.

network→get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

network→get_primaryDNS()

Returns the IP address of the primary name server to be used by the module.

network→get_readiness()

Returns the current established working mode of the network interface.

network→get_router()

Returns the IP address of the router on the device subnet (default gateway).

network→get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

network→get_subnetMask()

Returns the subnet mask currently used by the device.

network→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

network→get_userPassword()

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

network→get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→isOnline()

Checks if the network interface is currently reachable, without raising any error.

network→isOnline_async(**callback**, **context**)

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

network→load(**msValidity**)

Preloads the network interface cache with a specified validity duration.

network→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

network→load_async(**msValidity**, **callback**, **context**)

Preloads the network interface cache with a specified validity duration (asynchronous version).

network→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

network→nextNetwork()

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

network→ping(**host**)

Pings host to test the network connectivity.

network→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

network→set_adminPassword(**newval**)

Changes the password for the "admin" user.

network→set_callbackCredentials(**newval**)

Changes the credentials required to connect to the callback address.

network→set_callbackEncoding(**newval**)

Changes the encoding standard to use for representing notification values.

network→set_callbackInitialDelay(**newval**)

Changes the initial waiting time before first callback notifications, in seconds.

network→set_callbackMaxDelay(**newval**)

Changes the waiting time between two HTTP callbacks when there is nothing new.

network→set_callbackMethod(**newval**)

Changes the HTTP method used to notify callbacks for significant state changes.

network→set_callbackMinDelay(**newval**)

Changes the minimum waiting time between two HTTP callbacks, in seconds.

network→set_callbackSchedule(**newval**)

Changes the HTTP callback schedule strategy, as a text string.

network→set_callbackUrl(**newval**)

Changes the callback URL to notify significant state changes.

network→set_defaultPage(**newval**)

Changes the default HTML page returned by the hub.

network→set_discoverable(**newval**)

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→set_httpPort(**newval**)

Changes the default HTML page returned by the hub.

network→set_logicalName(**newval**)

Changes the logical name of the network interface.

network→set_ntpServer(**newval**)

Changes the IP address of the NTP server to be used by the module.

network→set_periodicCallbackSchedule(**interval**, **offset**)

Setup periodic HTTP callbacks (simplifed function).

network→set_primaryDNS(**newval**)

Changes the IP address of the primary name server to be used by the module.

network→set_secondaryDNS(**newval**)

Changes the IP address of the secondary name server to be used by the module.

network→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

network→set_userPassword(**newval**)

Changes the password for the "user" user.

network→set_wwwWatchdogDelay(**newval**)

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→triggerCallback()

Trigger an HTTP callback quickly.

network→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

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→useDHCPauto()

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

network→useStaticIP(**ipAddress**, **subnetMaskLen**, **router**)

Changes the configuration of the network interface to use a static IP address.

network→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YNetwork.FindNetwork() YNetwork
yFindNetwork()yFindNetwork()

Retrieves a network interface for a given identifier.

```
function FindNetwork( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the network interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YNetwork.isOnline()` to test if the network interface is indeed online at a given time. In case of ambiguity when looking for a network interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the network interface

Returns :

a `YNetwork` object allowing you to drive the network interface.

YNetwork.FindNetworkInContext() YNetwork
yFindNetworkInContext()

Retrieves a network interface for a given identifier in a YAPI context.

function **FindNetworkInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the network interface

Returns :

a `YNetwork` object allowing you to drive the network interface.

YNetwork.FirstNetwork() YNetwork
yFirstNetwork()yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

function **FirstNetwork**( )

Use the method `YNetwork.nextNetwork()` to iterate on next network interfaces.

Returns :

a pointer to a `YNetwork` object, corresponding to the first network interface currently online, or a `null` pointer if there are none.

YNetwork.FirstNetworkInContext() YNetwork
yFirstNetworkInContext()

Starts the enumeration of network interfaces currently accessible.

function **FirstNetworkInContext**(**yctx**)

Use the method `YNetwork.nextNetwork()` to iterate on next network interfaces.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YNetwork` object, corresponding to the first network interface currently online, or a `null` pointer if there are none.

network→callbackLogin()network.callbackLogin() YNetwork

Connects to the notification callback and saves the credentials required to log into it.

function **callbackLogin**(**username**, **password**)

The password is not stored into the module, only a hashed copy of the credentials are saved. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

username username required to log to the callback

password password required to log to the callback

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→clearCache()network.clearCache() YNetwork

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the network interface attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

network→describe()network.describe() YNetwork

Returns a short text that describes unambiguously the instance of the network interface in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the network interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

network→get_adminPassword() YNetwork
network→adminPassword()
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() YNetwork
network→advertisedValue()
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() YNetwork
network→callbackCredentials()
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_callbackEncoding() YNetwork
network→callbackEncoding()
network.get_callbackEncoding()

Returns the encoding standard to use for representing notification values.

function **get_callbackEncoding**( )

Returns :

a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV, Y_CALLBACKENCODING_YOCTO_API, Y_CALLBACKENCODING_JSON_NUM, Y_CALLBACKENCODING_EMONCMS, Y_CALLBACKENCODING_AZURE, Y_CALLBACKENCODING_INFLUXDB, Y_CALLBACKENCODING_MQTT and Y_CALLBACKENCODING_YOCTO_API_JZON corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns Y_CALLBACKENCODING_INVALID.

network→get_callbackInitialDelay() YNetwork
network→callbackInitialDelay()
network.get_callbackInitialDelay()

Returns the initial waiting time before first callback notifications, in seconds.

function **get_callbackInitialDelay**( )

Returns :
an integer corresponding to the initial waiting time before first callback notifications, in seconds

On failure, throws an exception or returns `Y_CALLBACKINITIALDELAY_INVALID`.

network→get_callbackMaxDelay() YNetwork
network→callbackMaxDelay()
network.get_callbackMaxDelay()

Returns the waiting time between two HTTP callbacks when there is nothing new.

function **get_callbackMaxDelay**( )

Returns :
an integer corresponding to the waiting time between two HTTP callbacks when there is nothing new

On failure, throws an exception or returns `Y_CALLBACKMAXDELAY_INVALID`.

network→get_callbackMethod() YNetwork
network→callbackMethod()
network.get_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

function **get_callbackMethod**( )

Returns :

a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns `Y_CALLBACKMETHOD_INVALID`.

network→get_callbackMinDelay() YNetwork
network→callbackMinDelay()
network.get_callbackMinDelay()

Returns the minimum waiting time between two HTTP callbacks, in seconds.

function **get_callbackMinDelay**( )

Returns :
an integer corresponding to the minimum waiting time between two HTTP callbacks, in seconds

On failure, throws an exception or returns `Y_CALLBACKMINDELAY_INVALID`.

network→get_callbackSchedule() YNetwork
network→callbackSchedule()
network.get_callbackSchedule()

Returns the HTTP callback schedule strategy, as a text string.

function **get_callbackSchedule**( )

Returns :
a string corresponding to the HTTP callback schedule strategy, as a text string

On failure, throws an exception or returns `Y_CALLBACKSCHEDULE_INVALID`.

network→get_callbackUrl() YNetwork
network→callbackUrl()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_defaultPage() YNetwork
network→defaultPage()network.get_defaultPage()

Returns the HTML page to serve for the URL "/"" of the hub.

function **get_defaultPage**( )

Returns :
a string corresponding to the HTML page to serve for the URL "/"" of the hub

On failure, throws an exception or returns `Y_DEFAULTPAGE_INVALID`.

network→get_discoverable() YNetwork
network→discoverable()network.get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

function **get_discoverable**( )

Returns :

either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns Y_DISCOVERABLE_INVALID.

network→get_errorMessage() YNetwork
network→errorMessage()
network.get_errorMessage()

Returns the error message of the latest error with the network interface.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the network interface object

network→get_errorType() YNetwork
network→errorType()network.get_errorType()

Returns the numerical error code of the latest error with the network interface.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the network interface object

network→get_friendlyName() YNetwork
network→friendlyName()network.get_friendlyName()

Returns a global identifier of the network interface in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the network interface if they are defined, otherwise the serial number of the module and the hardware identifier of the network interface (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the network interface using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

network→get_functionDescriptor() YNetwork
network→functionDescriptor()
network.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

network→get_functionId() YNetwork
network→functionId()network.get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the network interface (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

network→get_hardwareId() YNetwork
network→hardwareId()network.get_hardwareId()

Returns the unique hardware identifier of the network interface in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the network interface (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the network interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

network→get_httpPort() YNetwork
network→httpPort()network.get_httpPort()

Returns the HTML page to serve for the URL "/"" of the hub.

function **get_httpPort**( )

Returns :
an integer corresponding to the HTML page to serve for the URL "/"" of the hub

On failure, throws an exception or returns `Y_HTTPPORT_INVALID`.

network→get_ipAddress() YNetwork
network→ipAddress()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_ipConfig() YNetwork
network→ipConfig()network.get_ipConfig()

Returns the IP configuration of the network interface.

function **get_ipConfig**( )

If the network interface is setup to use a static IP address, the string starts with "STATIC:" and is followed by three parameters, separated by "/". The first is the device IP address, followed by the subnet mask length, and finally the router IP address (default gateway). For instance: "STATIC:192.168.1.14/16/192.168.1.1"

If the network interface is configured to receive its IP from a DHCP server, the string start with "DHCP:" and is followed by three parameters separated by "/". The first is the fallback IP address, then the fallback subnet mask length and finally the fallback router IP address. These three parameters are used when no DHCP reply is received.

Returns :
a string corresponding to the IP configuration of the network interface

On failure, throws an exception or returns `Y_IPCONFIG_INVALID`.

network→get_logicalName() YNetwork
network→logicalName()network.get_logicalName()

Returns the logical name of the network interface.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the network interface.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

network→get_macAddress() YNetwork
network→macAddress()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() YNetwork
network→module()network.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

network→get_ntpServer() YNetwork
network→ntpServer()network.get_ntpServer()

Returns the IP address of the NTP server to be used by the device.

function **get_ntpServer**( )

Returns :
a string corresponding to the IP address of the NTP server to be used by the device

On failure, throws an exception or returns `Y_NTPSERVER_INVALID`.

network→get_poeCurrent() YNetwork
network→poeCurrent()network.get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

function **get_poeCurrent**( )

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

Returns :
an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns `Y_POECURRENT_INVALID`.

network→get_primaryDNS() YNetwork
network→primaryDNS()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() YNetwork
network→readiness()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 at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS_4) is reached when the DNS server is reachable on the network. Level 5 (WWW_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

Returns :

a value among `Y_READINESS_DOWN`, `Y_READINESS_EXISTS`, `Y_READINESS_LINKED`, `Y_READINESS_LAN_OK` and `Y_READINESS_WWW_OK` corresponding to the current established working mode of the network interface

On failure, throws an exception or returns `Y_READINESS_INVALID`.

network→get_router() YNetwork
network→router()network.get_router()

Returns the IP address of the router on the device subnet (default gateway).

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() YNetwork
network→secondaryDNS()
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() YNetwork
network→subnetMask()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() YNetwork
network→userData()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() YNetwork
network→userPassword()
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→get_wwwWatchdogDelay() YNetwork
network→wwwWatchdogDelay()
network.get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

function **get_wwwWatchdogDelay**( )

A zero value disables automated reboot in case of Internet connectivity loss.

Returns :

an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

On failure, throws an exception or returns `Y_WWWWATCHDOGDELAY_INVALID`.

network→isOnline()network.isOnline() YNetwork

Checks if the network interface is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

Returns :

`true` if the network interface can be reached, and `false` otherwise

network→load()network.load() YNetwork

Preloads the network interface cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()network.loadAttribute() YNetwork

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

network→muteValueCallbacks() YNetwork
network.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→nextNetwork()network.nextNetwork() YNetwork

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

function **nextNetwork**( )

Returns :

a pointer to a `YNetwork` object, corresponding to a network interface currently online, or a `null` pointer if there are no more network interfaces to enumerate.

network→ping()network.ping() YNetwork

Pings host to test the network connectivity.

function **ping**(**host**)

Sends four ICMP ECHO_REQUEST requests from the module to the target host. This method returns a string with the result of the 4 ICMP ECHO_REQUEST requests.

Parameters :

host the hostname or the IP address of the target

Returns :

a string with the result of the ping.

network→registerValueCallback() YNetwork
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() YNetwork
network→setAdminPassword()
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() YNetwork
network→setCallbackCredentials()
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 the authentication scheme implemented by the callback, For Basic authentication, the hash is the MD5 of the string `username:password`. For Digest authentication, the hash is the MD5 of the string `username:realm:password`. For a simpler way to configure callback credentials, use function `callbackLogin` instead. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the credentials required to connect to the callback address

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackEncoding() YNetwork
network→setCallbackEncoding()
network.set_callbackEncoding()

Changes the encoding standard to use for representing notification values.

function **set_callbackEncoding**(**newval**)

Parameters :

newval a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV`, `Y_CALLBACKENCODING_YOCTO_API`, `Y_CALLBACKENCODING_JSON_NUM`, `Y_CALLBACKENCODING_EMONCMS`, `Y_CALLBACKENCODING_AZURE`, `Y_CALLBACKENCODING_INFLUXDB`, `Y_CALLBACKENCODING_MQTT` and `Y_CALLBACKENCODING_YOCTO_API_JZON` corresponding to the encoding standard to use for representing notification values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackInitialDelay() YNetwork
network→setCallbackInitialDelay()
network.set_callbackInitialDelay()

Changes the initial waiting time before first callback notifications, in seconds.

function **set_callbackInitialDelay**(**newval**)

Parameters :

newval an integer corresponding to the initial waiting time before first callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMaxDelay() YNetwork
network→setCallbackMaxDelay()
network.set_callbackMaxDelay()

Changes the waiting time between two HTTP callbacks when there is nothing new.

function **set_callbackMaxDelay**(**newval**)

Parameters :

newval an integer corresponding to the waiting time between two HTTP callbacks when there is nothing new

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMethod() YNetwork
network→setCallbackMethod()
network.set_callbackMethod()

Changes the HTTP method used to notify callbacks for significant state changes.

function **set_callbackMethod**(**newval**)

Parameters :

newval a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMinDelay() YNetwork
network→setCallbackMinDelay()
network.set_callbackMinDelay()

Changes the minimum waiting time between two HTTP callbacks, in seconds.

function **set_callbackMinDelay**(**newval**)

Parameters :

newval an integer corresponding to the minimum waiting time between two HTTP callbacks, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackSchedule() YNetwork
network→setCallbackSchedule()
network.set_callbackSchedule()

Changes the HTTP callback schedule strategy, as a text string.

function **set_callbackSchedule**(**newval**)

Parameters :

newval a string corresponding to the HTTP callback schedule strategy, as a text string

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackUrl() YNetwork
network→setCallbackUrl()network.set_callbackUrl()

Changes the callback URL to notify significant state changes.

function **set_callbackUrl**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the callback URL to notify significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_defaultPage() YNetwork
network→setDefaultPage()network.set_defaultPage()

Changes the default HTML page returned by the hub.

```
function set_defaultPage( newval)
```

If not value are set the hub return "index.html" which is the web interface of the hub. It is possible de change this page for file that has been uploaded on the hub.

Parameters :

newval a string corresponding to the default HTML page returned by the hub

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_discoverable() YNetwork
network→setDiscoverable()
network.set_discoverable()

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

function **set_discoverable**(**newval**)

Parameters :

newval either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_httpPort() YNetwork
network→setHttpPort()network.set_httpPort()

Changes the default HTML page returned by the hub.

function **set_httpPort**(**newval**)

If not value are set the hub return "index.html" which is the web interface of the hub. It is possible de change this page for file that has been uploaded on the hub.

Parameters :

newval an integer corresponding to the default HTML page returned by the hub

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_logicalName() YNetwork
network→setLogicalName()
network.set_logicalName()

Changes the logical name of the network interface.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the network interface.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_ntpServer() YNetwork
network→setNtpServer()network.set_ntpServer()

Changes the IP address of the NTP server to be used by the module.

function **set_ntpServer**(**newval**)

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 NTP 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_periodicCallbackSchedule() YNetwork
network→setPeriodicCallbackSchedule()
network.set_periodicCallbackSchedule()

Setup periodic HTTP callbacks (simplifed function).

function **set_periodicCallbackSchedule**(**interval**, **offset**)

Parameters :

interval a string representing the callback periodicity, expressed in seconds, minutes or hours, eg. "60s", "5m", "1h", "48h".

offset an integer representing the time offset relative to the period when the callback should occur. For instance, if the periodicity is 24h, an offset of 7 will make the callback occur each day at 7AM.

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_primaryDNS() YNetwork
network→setPrimaryDNS()network.set_primaryDNS()

Changes the IP address of the primary name server to be used by the module.

function **set_primaryDNS**(**newval**)

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the primary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_secondaryDNS() YNetwork
network→setSecondaryDNS()
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 overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the secondary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_userData() YNetwork
network→setUserData()network.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

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() YNetwork
network→setUserPassword()
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→set_wwwWatchdogDelay() YNetwork
network→setWwwWatchdogDelay()
network.set_wwwWatchdogDelay()

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

function **set_wwwWatchdogDelay**(**newval**)

A zero value disables automated reboot in case of Internet connectivity loss. The smallest valid non-zero timeout is 90 seconds.

Parameters :

newval an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→triggerCallback()network.triggerCallback() YNetwork

Trigger an HTTP callback quickly.

function **triggerCallback**( )

This function can even be called within an HTTP callback, in which case the next callback will be triggered 5 seconds after the end of the current callback, regardless if the minimum time between callbacks configured in the device.

Returns :
YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→unmuteValueCallbacks() YNetwork
network.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useDHCP()network.useDHCP() YNetwork

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

function **useDHCP**(**fallbackIpAddr**, **fallbackSubnetMaskLen**, **fallbackRouter**)

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

fallbackIpAddr	fallback IP address, to be used when no DHCP reply is received
fallbackSubnetMaskLen	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
fallbackRouter	fallback router IP address, to be used when no DHCP reply is received

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useDHCPauto()network.useDHCPauto() YNetwork

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

function **useDHCPauto**( )

Until an address is received from a DHCP server, the module uses an IP of the network 169.254.0.0/16 (APIPA). Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useStaticIP()network.useStaticIP() YNetwork

Changes the configuration of the network interface to use a static IP address.

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` when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→wait_async()network.wait_async() YNetwork

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.44. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_oscontrol.js'></script>
cpp     #include "yocto_oscontrol.h"
m       #import "yocto_oscontrol.h"
pas     uses yocto_oscontrol;
vb      yocto_oscontrol.vb
cs      yocto_oscontrol.cs
java    import com.yoctopuce.YoctoAPI.YOsControl;
uwp     import com.yoctopuce.YoctoAPI.YOsControl;
py      from yocto_oscontrol import *
php     require_once('yocto_oscontrol.php');
es      in HTML: <script src="../../lib/yocto_oscontrol.js"></script>
        in node.js: require('yoctolib-es2017/yocto_oscontrol.js');
```

Global functions
yFindOsControl(**func**) Retrieves OS control for a given identifier.
yFindOsControlInContext(**yctx**, **func**) Retrieves OS control for a given identifier in a YAPI context.
yFirstOsControl() Starts the enumeration of OS control currently accessible.
yFirstOsControlInContext(**yctx**) Starts the enumeration of OS control currently accessible.
`YOsControl` methods
oscontrol→clearCache() Invalidates the cache.
oscontrol→describe() Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
oscontrol→get_advertisedValue() Returns the current value of the OS control (no more than 6 characters).
oscontrol→get_errorMessage() Returns the error message of the latest error with the OS control.
oscontrol→get_errorType() Returns the numerical error code of the latest error with the OS control.
oscontrol→get_friendlyName() Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.
oscontrol→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
oscontrol→get_functionId() Returns the hardware identifier of the OS control, without reference to the module.
oscontrol→get_hardwareId()

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

oscontrol→get_logicalName()

Returns the logical name of the OS control.

oscontrol→get_module()

Gets the `YModule` object for the device on which the function is located.

oscontrol→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

oscontrol→get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

oscontrol→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

oscontrol→isOnline()

Checks if the OS control is currently reachable, without raising any error.

oscontrol→isOnline_async(**callback**, **context**)

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

oscontrol→load(**msValidity**)

Preloads the OS control cache with a specified validity duration.

oscontrol→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

oscontrol→load_async(**msValidity**, **callback**, **context**)

Preloads the OS control cache with a specified validity duration (asynchronous version).

oscontrol→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

oscontrol→nextOsControl()

Continues the enumeration of OS control started using `yFirstOsControl()`.

oscontrol→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

oscontrol→set_logicalName(**newval**)

Changes the logical name of the OS control.

oscontrol→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

oscontrol→shutdown(**secBeforeShutDown**)

Schedules an OS shutdown after a given number of seconds.

oscontrol→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

oscontrol→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YOsControl.FindOsControl() YOsControl
yFindOsControl()yFindOsControl()

Retrieves OS control for a given identifier.

function **FindOsControl**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YOsControl.isOnline()` to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the OS control

Returns :

a `YOsControl` object allowing you to drive the OS control.

YOsControl.FindOsControlInContext() YOsControl yFindOsControlInContext()

Retrieves OS control for a given identifier in a YAPI context.

function **FindOsControlInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YOsControl.isOnline()` to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the OS control

Returns :

a `YOsControl` object allowing you to drive the OS control.

YOsControl.FirstOsControl() YOsControl
yFirstOsControl()yFirstOsControl()

Starts the enumeration of OS control currently accessible.

function **FirstOsControl**( )

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

Returns :

a pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a `null` pointer if there are none.

YOsControl.FirstOsControlInContext() YOsControl yFirstOsControlInContext()

Starts the enumeration of OS control currently accessible.

function **FirstOsControlInContext**(**yctx**)

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a `null` pointer if there are none.

oscontrol→clearCache()oscontrol.clearCache() YOsControl

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the OS control attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

oscontrol→describe()oscontrol.describe() YOsControl

Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the OS control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

oscontrol→get_advertisedValue() YOsControl
oscontrol→advertisedValue()
oscontrol.get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the OS control (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

oscontrol→get_errorMessage() YOsControl
oscontrol→errorMessage()
oscontrol.get_errorMessage()

Returns the error message of the latest error with the OS control.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the OS control object

oscontrol→get_errorType() YOsControl
oscontrol→errorType()oscontrol.get_errorType()

Returns the numerical error code of the latest error with the OS control.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the OS control object

oscontrol→get_friendlyName() YOsControl
oscontrol→friendlyName()
oscontrol.get_friendlyName()

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the OS control if they are defined, otherwise the serial number of the module and the hardware identifier of the OS control (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the OS control using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

oscontrol→get_functionDescriptor() YOsControl
oscontrol→functionDescriptor()
oscontrol.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

oscontrol→get_functionId() YOsControl
oscontrol→functionId()oscontrol.get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the OS control (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

oscontrol→get_hardwareId() YOsControl
oscontrol→hardwareId()oscontrol.get_hardwareId()

Returns the unique hardware identifier of the OS control in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the OS control (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the OS control (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

oscontrol→get_logicalName() YOsControl
oscontrol→logicalName()
oscontrol.get_logicalName()

Returns the logical name of the OS control.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the OS control.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

oscontrol→get_module() YOsControl
oscontrol→module()oscontrol.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

oscontrol→get_shutdownCountdown() YOsControl
oscontrol→shutdownCountdown()
oscontrol.get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

function **get_shutdownCountdown**( )

Returns :

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns `Y_SHUTDOWNCOUNTDOWN_INVALID`.

oscontrol→get_userData() YOsControl
oscontrol→userData()oscontrol.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.

oscontrol→isOnline()oscontrol.isOnline() YOsControl

Checks if the OS control is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

Returns :

`true` if the OS control can be reached, and `false` otherwise

oscontrol→load()oscontrol.load() YOsControl

Preloads the OS control cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

oscontrol→loadAttribute()oscontrol.loadAttribute() YOsControl

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

oscontrol→muteValueCallbacks() YOsControl
oscontrol.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

oscontrol→nextOsControl() YOsControl
oscontrol.nextOsControl()

Continues the enumeration of OS control started using `yFirstOsControl()`.

function **nextOsControl**( )

Returns :

a pointer to a `YOsControl` object, corresponding to OS control currently online, or a `null` pointer if there are no more OS control to enumerate.

oscontrol→registerValueCallback() YOsControl
oscontrol.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.

oscontrol→set_logicalName() YOsControl
oscontrol→setLogicalName()
oscontrol.set_logicalName()

Changes the logical name of the OS control.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the OS control.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

oscontrol→set_userData() YOsControl
oscontrol→setUserData()oscontrol.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

oscontrol→shutdown()oscontrol.shutdown() YOsControl

Schedules an OS shutdown after a given number of seconds.

function **shutdown**(**secBeforeShutDown**)

Parameters :

secBeforeShutDown number of seconds before shutdown

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

oscontrol→unmuteValueCallbacks() YOsControl
oscontrol.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

oscontrol→wait_async()oscontrol.wait_async() YOsControl

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.45. Power function interface

The Yoctopuce class YPower allows you to read and configure Yoctopuce power sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to access the energy counter and the power factor.

In order to use the functions described here, you should include:

```
js    <script type='text/javascript' src='yocto_power.js'></script>
cpp   #include "yocto_power.h"
m     #import "yocto_power.h"
pas   uses yocto_power;
vb    yocto_power.vb
cs    yocto_power.cs
java  import com.yoctopuce.YoctoAPI.YPower;
uwp   import com.yoctopuce.YoctoAPI.YPower;
py    from yocto_power import *
php   require_once('yocto_power.php');
es    in HTML: <script src="../../lib/yocto_power.js"></script>
      in node.js: require('yoctolib-es2017/yocto_power.js');
```

Global functions

yFindPower(**func**)

Retrieves a electrical power sensor for a given identifier.

yFindPowerInContext(**yctx**, **func**)

Retrieves a electrical power sensor for a given identifier in a YAPI context.

yFirstPower()

Starts the enumeration of electrical power sensors currently accessible.

yFirstPowerInContext(**yctx**)

Starts the enumeration of electrical power sensors currently accessible.

YPower methods

power→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

power→clearCache()

Invalidates the cache.

power→describe()

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

power→get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

power→get_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

power→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

power→get_currentValue()

Returns the current value of the electrical power, in Watt, as a floating point number.

power→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

power→get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

power→get_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

power→get_friendlyName()

Returns a global identifier of the electrical power sensor in the format MODULE_NAME.FUNCTION_NAME.

power→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

power→get_hardwareId()

Returns the unique hardware identifier of the electrical power sensor in the form SERIAL.FUNCTIONID.

power→get_highestValue()

Returns the maximal value observed for the electrical power since the device was started.

power→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→get_logicalName()

Returns the logical name of the electrical power sensor.

power→get_lowestValue()

Returns the minimal value observed for the electrical power since the device was started.

power→get_meter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

power→get_meterTimer()

Returns the elapsed time since last energy counter reset, in seconds.

power→get_module()

Gets the YModule object for the device on which the function is located.

power→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

power→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

power→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→get_resolution()

Returns the resolution of the measured values.

power→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

power→get_unit()

Returns the measuring unit for the electrical power.

power→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

power→isOnline()

Checks if the electrical power sensor is currently reachable, without raising any error.

power→isOnline_async(callback, context)

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

power→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

power→load(msValidity)

Preloads the electrical power sensor cache with a specified validity duration.

power→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

power→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

power→load_async(msValidity, callback, context)

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

power→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

power→nextPower()

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

power→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

power→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

power→reset()

Resets the energy counter.

power→set_highestValue(newval)

Changes the recorded maximal value observed.

power→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

power→set_logicalName(newval)

Changes the logical name of the electrical power sensor.

power→set_lowestValue(newval)

Changes the recorded minimal value observed.

power→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

power→set_resolution(newval)

Changes the resolution of the measured physical values.

power→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

power→startDataLogger()

Starts the data logger on the device.

power→stopDataLogger()

Stops the datalogger on the device.

power→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

power→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPower.FindPower() YPower
yFindPower()yFindPower()

Retrieves a electrical power sensor for a given identifier.

function **FindPower**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the electrical power sensor

Returns :

a `YPower` object allowing you to drive the electrical power sensor.

YPower.FindPowerInContext() YPower yFindPowerInContext()

Retrieves a electrical power sensor for a given identifier in a YAPI context.

function **FindPowerInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the electrical power sensor

Returns :

a `YPower` object allowing you to drive the electrical power sensor.

YPower.FirstPower() YPower yFirstPower()yFirstPower()

Starts the enumeration of electrical power sensors currently accessible.

function **FirstPower**( )

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

Returns :

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

YPower.FirstPowerInContext() YPower
yFirstPowerInContext()

Starts the enumeration of electrical power sensors currently accessible.

function **FirstPowerInContext**(**yctx**)

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

power→calibrateFromPoints() YPower
power.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→clearCache()power.clearCache() YPower

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the electrical power sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

power→describe()power.describe() YPower

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the electrical power sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

power→get_advertisedValue() YPower
power→advertisedValue()
power.get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the electrical power sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

power→get_cosPhi() YPower
power→cosPhi()power.get_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

function **get_cosPhi**( )

Returns :

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns `Y_COSPHI_INVALID`.

power→get_currentRawValue() YPower
power→currentRawValue()
power.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

power→get_currentValue() YPower
power→currentValue()power.get_currentValue()

Returns the current value of the electrical power, in Watt, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the electrical power, in Watt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

power→get_dataLogger() YPower
power→dataLogger()power.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

power→get_errorMessage() YPower power→errorMessage()power.get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the electrical power sensor object

power→get_errorType() YPower
power→errorType()power.get_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the electrical power sensor object

power→get_friendlyName() YPower
power→friendlyName()power.get_friendlyName()

Returns a global identifier of the electrical power sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the electrical power sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

power→get_functionDescriptor() YPower
power→functionDescriptor()
power.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

power→get_functionId() YPower
power→functionId()power.get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the electrical power sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

power→get_hardwareId() YPower
power→hardwareId()power.get_hardwareId()

Returns the unique hardware identifier of the electrical power sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the electrical power sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the electrical power sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

power→get_highestValue() YPower
power→highestValue()power.get_highestValue()

Returns the maximal value observed for the electrical power since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the electrical power since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

power→get_logFrequency() YPower
power→logFrequency()power.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

power→get_logicalName() YPower
power→logicalName()power.get_logicalName()

Returns the logical name of the electrical power sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the electrical power sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

power→get_lowestValue() YPower
power→lowestValue()power.get_lowestValue()

Returns the minimal value observed for the electrical power since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the electrical power since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

power→get_meter() YPower
power→meter()power.get_meter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

function **get_meter**( )

Note that this counter is reset at each start of the device.

Returns :
a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns `Y_METER_INVALID`.

power→get_meterTimer() YPower
power→meterTimer()power.get_meterTimer()

Returns the elapsed time since last energy counter reset, in seconds.

function **get_meterTimer**( )

Returns :
an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns `Y_METERTIMER_INVALID`.

power→get_module() YPower
power→module()power.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

power→get_recordedData() YPower
power→recordedData()power.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

power→get_reportFrequency() YPower
power→reportFrequency()
power.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

power→get_resolution() YPower
power→resolution()power.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

power→get_sensorState() YPower
power→sensorState()power.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

power→get_unit() YPower
power→unit()power.get_unit()

Returns the measuring unit for the electrical power.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns `Y_UNIT_INVALID`.

power→get_userData() YPower
power→userData()power.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.

power→isOnline()power.isOnline() YPower

Checks if the electrical power sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

Returns :

`true` if the electrical power sensor can be reached, and `false` otherwise

power→load()power.load() YPower

Preloads the electrical power sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

power→loadAttribute()power.loadAttribute() YPower

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

power→loadCalibrationPoints() YPower
power.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→muteValueCallbacks() YPower
power.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

power→nextPower()power.nextPower() YPower

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

function **nextPower**( )

Returns :

a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a `null` pointer if there are no more electrical power sensors to enumerate.

power→registerTimedReportCallback() YPower
power.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

power→registerValueCallback() YPower
power.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.

power→reset()power.reset() YPower

Resets the energy counter.

function **reset**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_highestValue() YPower
power→setHighestValue()power.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.

power→set_logFrequency() YPower
power→setLogFrequency()power.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logicalName() YPower
power→setLogicalName()power.set_logicalName()

Changes the logical name of the electrical power sensor.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the electrical power sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_lowestValue() YPower
power→setLowestValue()power.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.

power→set_reportFrequency() YPower
power→setReportFrequency()
power.set_reportFrequency()

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_resolution() YPower
power→setResolution()power.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_userData() YPower
power→setUserData()power.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

power→startDataLogger()power.startDataLogger() YPower

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

power→stopDataLogger()power.stopDataLogger() YPower

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

power→unmuteValueCallbacks() YPower
power.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

power→wait_async()power.wait_async() YPower

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.46. External power supply control interface

Yoctopuce application programming interface allows you to control the power ouput featured on some devices such as the Yocto-Serial.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_poweroutput.js'></script>
cpp     #include "yocto_poweroutput.h"
m       #import "yocto_poweroutput.h"
pas     uses yocto_poweroutput;
vb      yocto_poweroutput.vb
cs      yocto_poweroutput.cs
java    import com.yoctopuce.YoctoAPI.YPowerOutput;
uwp     import com.yoctopuce.YoctoAPI.YPowerOutput;
py      from yocto_poweroutput import *
php     require_once('yocto_poweroutput.php');
es      in HTML: <script src="../../lib/yocto_poweroutput.js"></script>
        in node.js: require('yoctolib-es2017/yocto_poweroutput.js');
```

Global functions

yFindPowerOutput(**func**)

Retrieves a dual power ouput control for a given identifier.

yFindPowerOutputInContext(**yctx**, **func**)

Retrieves a dual power ouput control for a given identifier in a YAPI context.

yFirstPowerOutput()

Starts the enumeration of dual power ouput controls currently accessible.

yFirstPowerOutputInContext(**yctx**)

Starts the enumeration of dual power ouput controls currently accessible.

`YPowerOutput` methods

poweroutput→clearCache()

Invalidates the cache.

poweroutput→describe()

Returns a short text that describes unambiguously the instance of the power ouput control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

poweroutput→get_advertisedValue()

Returns the current value of the power ouput control (no more than 6 characters).

poweroutput→get_errorMessage()

Returns the error message of the latest error with the power ouput control.

poweroutput→get_errorType()

Returns the numerical error code of the latest error with the power ouput control.

poweroutput→get_friendlyName()

Returns a global identifier of the power ouput control in the format `MODULE_NAME.FUNCTION_NAME`.

poweroutput→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

poweroutput→get_functionId()

Returns the hardware identifier of the power ouput control, without reference to the module.

poweroutput→get_hardwareId()

Returns the unique hardware identifier of the power ouput control in the form `SERIAL.FUNCTIONID`.

poweroutput→get_logicalName()

Returns the logical name of the power ouput control.

poweroutput→get_module()

Gets the `YModule` object for the device on which the function is located.

poweroutput→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

poweroutput→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

poweroutput→get_voltage()

Returns the voltage on the power output featured by the module.

poweroutput→isOnline()

Checks if the power ouput control is currently reachable, without raising any error.

poweroutput→isOnline_async(callback, context)

Checks if the power ouput control is currently reachable, without raising any error (asynchronous version).

poweroutput→load(msValidity)

Preloads the power ouput control cache with a specified validity duration.

poweroutput→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

poweroutput→load_async(msValidity, callback, context)

Preloads the power ouput control cache with a specified validity duration (asynchronous version).

poweroutput→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

poweroutput→nextPowerOutput()

Continues the enumeration of dual power ouput controls started using `yFirstPowerOutput()`.

poweroutput→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

poweroutput→set_logicalName(newval)

Changes the logical name of the power ouput control.

poweroutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

poweroutput→set_voltage(newval)

Changes the voltage on the power output provided by the module.

poweroutput→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

poweroutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPowerOutput.FindPowerOutput() YPowerOutput yFindPowerOutput()yFindPowerOutput()

Retrieves a dual power ouput control for a given identifier.

```
function FindPowerOutput( 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 ouput control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPowerOutput.isOnline()` to test if the power ouput control is indeed online at a given time. In case of ambiguity when looking for a dual power ouput 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the power ouput control

Returns :

a `YPowerOutput` object allowing you to drive the power ouput control.

YPowerOutput.FindPowerOutputInContext() YPowerOutput yFindPowerOutputInContext()

Retrieves a dual power ouput control for a given identifier in a YAPI context.

function **FindPowerOutputInContext**(**yctx**, **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 ouput control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPowerOutput.isOnline()` to test if the power ouput control is indeed online at a given time. In case of ambiguity when looking for a dual power ouput 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 :

yctx a YAPI context

func a string that uniquely characterizes the power ouput control

Returns :

a `YPowerOutput` object allowing you to drive the power ouput control.

YPowerOutput.FirstPowerOutput() YPowerOutput yFirstPowerOutput()yFirstPowerOutput()

Starts the enumeration of dual power ouput controls currently accessible.

function **FirstPowerOutput**( )

Use the method `YPowerOutput.nextPowerOutput()` to iterate on next dual power ouput controls.

Returns :

a pointer to a `YPowerOutput` object, corresponding to the first dual power ouput control currently online, or a `null` pointer if there are none.

YPowerOutput.FirstPowerOutputInContext() YPowerOutput yFirstPowerOutputInContext()

Starts the enumeration of dual power ouput controls currently accessible.

function **FirstPowerOutputInContext**(**yctx**)

Use the method `YPowerOutput.nextPowerOutput()` to iterate on next dual power ouput controls.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YPowerOutput` object, corresponding to the first dual power ouput control currently online, or a `null` pointer if there are none.

poweroutput→clearCache()poweroutput.clearCache() YPowerOutput

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the power ouput control attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

poweroutput→describe()poweroutput.describe() YPowerOutput

Returns a short text that describes unambiguously the instance of the power ouput control in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the power ouput control (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

poweroutput→get_advertisedValue() YPowerOutput
poweroutput→advertisedValue()
poweroutput.get_advertisedValue()

Returns the current value of the power ouput control (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the power ouput control (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

poweroutput→get_errorMessage() YPowerOutput
poweroutput→errorMessage()
poweroutput.get_errorMessage()

Returns the error message of the latest error with the power ouput control.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the power ouput control object

poweroutput→get_errorType() YPowerOutput
poweroutput→errorType()
poweroutput.get_errorType()

Returns the numerical error code of the latest error with the power ouput control.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the power ouput control object

poweroutput→get_friendlyName() YPowerOutput
poweroutput→friendlyName()
poweroutput.get_friendlyName()

Returns a global identifier of the power ouput control in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the power ouput control if they are defined, otherwise the serial number of the module and the hardware identifier of the power ouput control (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the power ouput control using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

poweroutput→get_functionDescriptor() YPowerOutput
poweroutput→functionDescriptor()
poweroutput.get_functionDescriptor()

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.

poweroutput→get_functionId() YPowerOutput
poweroutput→functionId()
poweroutput.get_functionId()

Returns the hardware identifier of the power ouput control, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the power ouput control (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

poweroutput→get_hardwareId() YPowerOutput
poweroutput→hardwareId()
poweroutput.get_hardwareId()

Returns the unique hardware identifier of the power ouput control in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the power ouput control (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the power ouput control (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

poweroutput→get_logicalName() YPowerOutput
poweroutput→logicalName()
poweroutput.get_logicalName()

Returns the logical name of the power ouput control.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the power ouput control.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

poweroutput→get_module() YPowerOutput
poweroutput→module()poweroutput.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

poweroutput→get_userData() YPowerOutput
poweroutput→userData()poweroutput.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.

poweroutput→get_voltage() YPowerOutput
poweroutput→voltage()poweroutput.get_voltage()

Returns the voltage on the power output featured by the module.

function **get_voltage**( )

Returns :

a value among `Y_VOLTAGE_OFF`, `Y_VOLTAGE_OUT3V3` and `Y_VOLTAGE_OUT5V` corresponding to the voltage on the power output featured by the module

On failure, throws an exception or returns `Y_VOLTAGE_INVALID`.

poweroutput→isOnline()poweroutput.isOnline() YPowerOutput

Checks if the power ouput control is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the power ouput control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power ouput control.

Returns :

`true` if the power ouput control can be reached, and `false` otherwise

poweroutput→load()poweroutput.load() YPowerOutput

Preloads the power ouput control cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

poweroutput→loadAttribute() YPowerOutput
poweroutput.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

poweroutput→muteValueCallbacks() YPowerOutput
poweroutput.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

poweroutput→nextPowerOutput() YPowerOutput
poweroutput.nextPowerOutput()

Continues the enumeration of dual power ouput controls started using `yFirstPowerOutput()`.

function **nextPowerOutput**( )

Returns :

a pointer to a `YPowerOutput` object, corresponding to a dual power ouput control currently online, or a `null` pointer if there are no more dual power ouput controls to enumerate.

poweroutput→registerValueCallback() YPowerOutput
poweroutput.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.

poweroutput→set_logicalName() YPowerOutput
poweroutput→setLogicalName()
poweroutput.set_logicalName()

Changes the logical name of the power ouput 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 ouput control.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

poweroutput→set_userData() YPowerOutput
poweroutput→setUserData()
poweroutput.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

poweroutput→set_voltage() YPowerOutput
poweroutput→setVoltage()poweroutput.set_voltage()

Changes the voltage on the power output provided by the module.

function **set_voltage**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_VOLTAGE_OFF`, `Y_VOLTAGE_OUT3V3` and `Y_VOLTAGE_OUT5V` corresponding to the voltage on the power output provided by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

poweroutput→unmuteValueCallbacks() YPowerOutput
poweroutput.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

poweroutput→wait_async()poweroutput.wait_async() YPowerOutput

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.47. Pressure function interface

The Yoctopuce class YPressure allows you to read and configure Yoctopuce pressure sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pressure.js'></script>
cpp	#include "yocto_pressure.h"
m	#import "yocto_pressure.h"
pas	uses yocto_pressure;
vb	yocto_pressure.vb
cs	yocto_pressure.cs
java	import com.yoctopuce.YoctoAPI.YPressure;
uwp	import com.yoctopuce.YoctoAPI.YPressure;
py	from yocto_pressure import *
php	require_once('yocto_pressure.php');
es	in HTML: <script src="../../lib/yocto_pressure.js"></script> in node.js: require('yoctolib-es2017/yocto_pressure.js');

Global functions

yFindPressure(**func**)

Retrieves a pressure sensor for a given identifier.

yFindPressureInContext(**yctx**, **func**)

Retrieves a pressure sensor for a given identifier in a YAPI context.

yFirstPressure()

Starts the enumeration of pressure sensors currently accessible.

yFirstPressureInContext(**yctx**)

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→clearCache()

Invalidates the cache.

pressure→describe()

Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

pressure→get_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

pressure→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

pressure→get_currentValue()

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

pressure→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

pressure→get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

pressure→get_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

pressure→get_friendlyName()

Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.

pressure→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pressure→get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

pressure→get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

pressure→get_highestValue()

Returns the maximal value observed for the pressure since the device was started.

pressure→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pressure→get_logicalName()

Returns the logical name of the pressure sensor.

pressure→get_lowestValue()

Returns the minimal value observed for the pressure since the device was started.

pressure→get_module()

Gets the `YModule` object for the device on which the function is located.

pressure→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pressure→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pressure→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pressure→get_resolution()

Returns the resolution of the measured values.

pressure→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

pressure→get_unit()

Returns the measuring unit for the pressure.

pressure→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pressure→isOnline()

Checks if the pressure sensor is currently reachable, without raising any error.

pressure→isOnline_async(**callback**, **context**)

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

pressure→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

pressure→load(**msValidity**)

Preloads the pressure sensor cache with a specified validity duration.

pressure→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

pressure→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

pressure→load_async(msValidity, callback, context)

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

pressure→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

pressure→nextPressure()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

pressure→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

pressure→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pressure→set_highestValue(newval)

Changes the recorded maximal value observed.

pressure→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

pressure→set_logicalName(newval)

Changes the logical name of the pressure sensor.

pressure→set_lowestValue(newval)

Changes the recorded minimal value observed.

pressure→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

pressure→set_resolution(newval)

Changes the resolution of the measured physical values.

pressure→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pressure→startDataLogger()

Starts the data logger on the device.

pressure→stopDataLogger()

Stops the datalogger on the device.

pressure→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

pressure→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPressure.FindPressure() YPressure
yFindPressure()yFindPressure()

Retrieves a pressure sensor for a given identifier.

```
function FindPressure( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the pressure sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPressure.isOnline()` to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the pressure sensor

Returns :

a `YPressure` object allowing you to drive the pressure sensor.

YPressure.FindPressureInContext() YPressure yFindPressureInContext()

Retrieves a pressure sensor for a given identifier in a YAPI context.

function **FindPressureInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the pressure sensor

Returns :

a `YPressure` object allowing you to drive the pressure sensor.

YPressure.FirstPressure() YPressure yFirstPressure()yFirstPressure()

Starts the enumeration of pressure sensors currently accessible.

function **FirstPressure**( )

Use the method `YPressure.nextPressure()` to iterate on next pressure sensors.

Returns :

a pointer to a `YPressure` object, corresponding to the first pressure sensor currently online, or a `null` pointer if there are none.

YPressure.FirstPressureInContext() YPressure yFirstPressureInContext()

Starts the enumeration of pressure sensors currently accessible.

function **FirstPressureInContext**(**yctx**)

Use the method `YPressure.nextPressure()` to iterate on next pressure sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YPressure` object, corresponding to the first pressure sensor currently online, or a `null` pointer if there are none.

pressure→calibrateFromPoints() YPressure
pressure.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

function **calibrateFromPoints**(**rawValues**, **refValues**)

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→clearCache()pressure.clearCache() YPressure

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the pressure sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

pressure→describe()pressure.describe() YPressure

Returns a short text that describes unambiguously the instance of the pressure sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the pressure sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

pressure→get_advertisedValue() YPressure
pressure→advertisedValue()
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() YPressure
pressure→currentRawValue()
pressure.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

function **get_currentRawValue**( )

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

pressure→get_currentValue() YPressure
pressure→currentValue()pressure.get_currentValue()

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the pressure, in millibar (hPa), as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

pressure→get_dataLogger() YPressure
pressure→dataLogger()pressure.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

pressure→get_errorMessage() YPressure
pressure→errorMessage()
pressure.get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the pressure sensor object

pressure→get_errorType() YPressure
pressure→errorType()pressure.get_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the pressure sensor object

pressure→get_friendlyName() YPressure
pressure→friendlyName()
pressure.get_friendlyName()

Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the pressure sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the pressure sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the pressure sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pressure→get_functionDescriptor() YPressure
pressure→functionDescriptor()
pressure.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pressure→get_functionId() YPressure
pressure→functionId()pressure.get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the pressure sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pressure→get_hardwareId() YPressure
pressure→hardwareId()pressure.get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the pressure sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the pressure sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pressure→get_highestValue() YPressure
pressure→highestValue()
pressure.get_highestValue()

Returns the maximal value observed for the pressure since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the pressure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

pressure→get_logFrequency() YPressure
pressure→logFrequency()
pressure.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

pressure→get_logicalName() YPressure
pressure→logicalName()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() YPressure
pressure→lowestValue()pressure.get_lowestValue()

Returns the minimal value observed for the pressure since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the pressure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

pressure→get_module() YPressure
pressure→module()pressure.get_module()

Gets the YModule object for the device on which the function is located.

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_recordedData() YPressure
pressure→recordedData()
pressure.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pressure→get_reportFrequency() YPressure
pressure→reportFrequency()
pressure.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

pressure→get_resolution() YPressure
pressure→resolution()pressure.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

pressure→get_sensorState() YPressure
pressure→sensorState()pressure.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

pressure→get_unit() YPressure
pressure→unit()pressure.get_unit()

Returns the measuring unit for the pressure.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

pressure→get_userData() YPressure
pressure→userData()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()pressure.isOnline() YPressure

Checks if the pressure sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

Returns :

`true` if the pressure sensor can be reached, and `false` otherwise

pressure→load()pressure.load() YPressure

Preloads the pressure sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()pressure.loadAttribute() YPressure

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

pressure→loadCalibrationPoints() YPressure
pressure.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→muteValueCallbacks() YPressure
pressure.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→nextPressure()pressure.nextPressure() YPressure

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

function **nextPressure**( )

Returns :

a pointer to a `YPressure` object, corresponding to a pressure sensor currently online, or a `null` pointer if there are no more pressure sensors to enumerate.

pressure→registerTimedReportCallback() YPressure
pressure.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

pressure→registerValueCallback() YPressure
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() YPressure
pressure→setHighestValue()
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_logFrequency() YPressure
pressure→setLogFrequency()
pressure.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logicalName() YPressure
pressure→setLogicalName()
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() YPressure
pressure→setLowestValue()
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_reportFrequency() YPressure
pressure→setReportFrequency()
pressure.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_resolution() YPressure
pressure→setResolution()pressure.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_userData() YPressure
pressure→setUserData()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

pressure→startDataLogger() YPressure
pressure.startDataLogger()

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

pressure→stopDataLogger() YPressure
pressure.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

pressure→unmuteValueCallbacks() YPressure
pressure.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→wait_async()pressure.wait_async() YPressure

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.48. Proximity function interface

The Yoctopuce class YProximity allows you to use and configure Yoctopuce proximity sensors. It inherits from the YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to easily perform a one-point linear calibration to compensate the effect of a glass or filter placed in front of the sensor.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_proximity.js'></script>
cpp	#include "yocto_proximity.h"
m	#import "yocto_proximity.h"
pas	uses yocto_proximity;
vb	yocto_proximity.vb
cs	yocto_proximity.cs
java	import com.yoctopuce.YoctoAPI.YProximity;
uwp	import com.yoctopuce.YoctoAPI.YProximity;
py	from yocto_proximity import *
php	require_once('yocto_proximity.php');
es	in HTML: <script src="../../lib/yocto_proximity.js"></script> in node.js: require('yoctolib-es2017/yocto_proximity.js');

Global functions
yFindProximity(**func**) Retrieves a proximity sensor for a given identifier.
yFindProximityInContext(**yctx**, **func**) Retrieves a proximity sensor for a given identifier in a YAPI context.
yFirstProximity() Starts the enumeration of proximity sensors currently accessible.
yFirstProximityInContext(**yctx**) Starts the enumeration of proximity sensors currently accessible.
`YProximity` methods
proximity→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
proximity→clearCache() Invalidates the cache.
proximity→describe() Returns a short text that describes unambiguously the instance of the proximity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
proximity→get_advertisedValue() Returns the current value of the proximity sensor (no more than 6 characters).
proximity→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.
proximity→get_currentValue() Returns the current value of the proximity detection, in the specified unit, as a floating point number.
proximity→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
proximity→get_detectionThreshold()

Returns the threshold used to determine the logical state of the proximity sensor, when considered as a binary input (on/off).

proximity→get_errorMessage()

Returns the error message of the latest error with the proximity sensor.

proximity→get_errorType()

Returns the numerical error code of the latest error with the proximity sensor.

proximity→get_friendlyName()

Returns a global identifier of the proximity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

proximity→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

proximity→get_functionId()

Returns the hardware identifier of the proximity sensor, without reference to the module.

proximity→get_hardwareId()

Returns the unique hardware identifier of the proximity sensor in the form `SERIAL.FUNCTIONID`.

proximity→get_highestValue()

Returns the maximal value observed for the proximity detection since the device was started.

proximity→get_isPresent()

Returns true if the input (considered as binary) is active (detection value is smaller than the specified `threshold`), and false otherwise.

proximity→get_lastTimeApproached()

Returns the number of elapsed milliseconds between the module power on and the last observed detection (the input contact transitioned from absent to present).

proximity→get_lastTimeRemoved()

Returns the number of elapsed milliseconds between the module power on and the last observed detection (the input contact transitioned from present to absent).

proximity→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

proximity→get_logicalName()

Returns the logical name of the proximity sensor.

proximity→get_lowestValue()

Returns the minimal value observed for the proximity detection since the device was started.

proximity→get_module()

Gets the `YModule` object for the device on which the function is located.

proximity→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

proximity→get_proximityReportMode()

Returns the parameter (sensor value, presence or pulse count) returned by the get_currentValue function and callbacks.

proximity→get_pulseCounter()

Returns the pulse counter value.

proximity→get_pulseTimer()

Returns the timer of the pulse counter (ms).

proximity→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

proximity→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

proximity→get_resolution()

Returns the resolution of the measured values.

proximity→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

proximity→get_signalValue()

Returns the current value of signal measured by the proximity sensor.

proximity→get_unit()

Returns the measuring unit for the proximity detection.

proximity→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

proximity→isOnline()

Checks if the proximity sensor is currently reachable, without raising any error.

proximity→isOnline_async(**callback**, **context**)

Checks if the proximity sensor is currently reachable, without raising any error (asynchronous version).

proximity→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

proximity→load(**msValidity**)

Preloads the proximity sensor cache with a specified validity duration.

proximity→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

proximity→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

proximity→load_async(**msValidity**, **callback**, **context**)

Preloads the proximity sensor cache with a specified validity duration (asynchronous version).

proximity→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

proximity→nextProximity()

Continues the enumeration of proximity sensors started using `yFirstProximity()`.

proximity→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

proximity→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

proximity→resetCounter()

Resets the pulse counter value as well as its timer.

proximity→set_detectionThreshold(**newval**)

Changes the threshold used to determine the logical state of the proximity sensor, when considered as a binary input (on/off).

proximity→set_highestValue(**newval**)

Changes the recorded maximal value observed.

proximity→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

proximity→set_logicalName(**newval**)

Changes the logical name of the proximity sensor.

proximity→set_lowestValue(newval)

Changes the recorded minimal value observed.

proximity→set_proximityReportMode(newval)

Modifies the parameter type (sensor value, presence or pulse count) returned by the get_currentValue function and callbacks.

proximity→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

proximity→set_resolution(newval)

Changes the resolution of the measured physical values.

proximity→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

proximity→startDataLogger()

Starts the data logger on the device.

proximity→stopDataLogger()

Stops the datalogger on the device.

proximity→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

proximity→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YProximity.FindProximity() YProximity yFindProximity()yFindProximity()

Retrieves a proximity sensor for a given identifier.

```
function FindProximity( 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 proximity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YProximity.isOnline()` to test if the proximity sensor is indeed online at a given time. In case of ambiguity when looking for a proximity 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the proximity sensor

Returns :

a `YProximity` object allowing you to drive the proximity sensor.

YProximity.FindProximityInContext() YProximity yFindProximityInContext()

Retrieves a proximity sensor for a given identifier in a YAPI context.

function **FindProximityInContext**(**yctx**, **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 proximity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YProximity.isOnline()` to test if the proximity sensor is indeed online at a given time. In case of ambiguity when looking for a proximity 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 :

yctx a YAPI context

func a string that uniquely characterizes the proximity sensor

Returns :

a `YProximity` object allowing you to drive the proximity sensor.

YProximity.FirstProximity() YProximity yFirstProximity()yFirstProximity()

Starts the enumeration of proximity sensors currently accessible.

```
function FirstProximity( )
```

Use the method `YProximity.nextProximity()` to iterate on next proximity sensors.

Returns :

a pointer to a `YProximity` object, corresponding to the first proximity sensor currently online, or a `null` pointer if there are none.

YProximity.FirstProximityInContext() YProximity
yFirstProximityInContext()

Starts the enumeration of proximity sensors currently accessible.

function **FirstProximityInContext**(**yctx**)

Use the method `YProximity.nextProximity()` to iterate on next proximity sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YProximity` object, corresponding to the first proximity sensor currently online, or a `null` pointer if there are none.

proximity→calibrateFromPoints() YProximity
proximity.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.

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.

proximity→clearCache()proximity.clearCache() YProximity

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the proximity sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

proximity→describe()proximity.describe() YProximity

Returns a short text that describes unambiguously the instance of the proximity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the proximity sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

proximity→get_advertisedValue() YProximity
proximity→advertisedValue()
proximity.get_advertisedValue()

Returns the current value of the proximity sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the proximity sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

proximity→get_currentRawValue() YProximity
proximity→currentRawValue()
proximity.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

proximity→get_currentValue() YProximity
proximity→currentValue()
proximity.get_currentValue()

Returns the current value of the proximity detection, in the specified unit, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the proximity detection, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

proximity→get_dataLogger() YProximity
proximity→dataLogger()proximity.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

proximity→get_detectionThreshold() YProximity
proximity→detectionThreshold()
proximity.get_detectionThreshold()

Returns the threshold used to determine the logical state of the proximity sensor, when considered as a binary input (on/off).

function **get_detectionThreshold**( )

Returns :

an integer corresponding to the threshold used to determine the logical state of the proximity sensor, when considered as a binary input (on/off)

On failure, throws an exception or returns `Y_DETECTIONTHRESHOLD_INVALID`.

proximity→get_errorMessage() YProximity
proximity→errorMessage()
proximity.get_errorMessage()

Returns the error message of the latest error with the proximity sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the proximity sensor object

proximity→get_errorType() YProximity
proximity→errorType()proximity.get_errorType()

Returns the numerical error code of the latest error with the proximity sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the proximity sensor object

proximity→get_friendlyName() YProximity
proximity→friendlyName()
proximity.get_friendlyName()

Returns a global identifier of the proximity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the proximity sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the proximity sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the proximity sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

proximity→get_functionDescriptor() YProximity
proximity→functionDescriptor()
proximity.get_functionDescriptor()

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.

proximity→get_functionId() YProximity
proximity→functionId()proximity.get_functionId()

Returns the hardware identifier of the proximity sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the proximity sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

proximity→get_hardwareId() YProximity
proximity→hardwareId()proximity.get_hardwareId()

Returns the unique hardware identifier of the proximity sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the proximity sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the proximity sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

proximity→get_highestValue() YProximity
proximity→highestValue()
proximity.get_highestValue()

Returns the maximal value observed for the proximity detection since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the proximity detection since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

proximity→get_isPresent() YProximity
proximity→isPresent()proximity.get_isPresent()

Returns true if the input (considered as binary) is active (detection value is smaller than the specified `threshold`), and false otherwise.

function **get_isPresent**( )

Returns :

either `Y_ISPRESENT_FALSE` or `Y_ISPRESENT_TRUE`, according to true if the input (considered as binary) is active (detection value is smaller than the specified `threshold`), and false otherwise

On failure, throws an exception or returns `Y_ISPRESENT_INVALID`.

proximity→get_lastTimeApproached() YProximity
proximity→lastTimeApproached()
proximity.get_lastTimeApproached()

Returns the number of elapsed milliseconds between the module power on and the last observed detection (the input contact transitioned from absent to present).

function **get_lastTimeApproached**( )

Returns :

an integer corresponding to the number of elapsed milliseconds between the module power on and the last observed detection (the input contact transitioned from absent to present)

On failure, throws an exception or returns `Y_LASTTIMEAPPROACHED_INVALID`.

proximity→get_lastTimeRemoved() YProximity
proximity→lastTimeRemoved()
proximity.get_lastTimeRemoved()

Returns the number of elapsed milliseconds between the module power on and the last observed detection (the input contact transitioned from present to absent).

function **get_lastTimeRemoved**( )

Returns :

an integer corresponding to the number of elapsed milliseconds between the module power on and the last observed detection (the input contact transitioned from present to absent)

On failure, throws an exception or returns `Y_LASTTIMEREMOVED_INVALID`.

proximity→get_logFrequency() YProximity
proximity→logFrequency()
proximity.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

proximity→get_logicalName()
proximity→logicalName()
proximity.get_logicalName()

YProximity

Returns the logical name of the proximity sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the proximity sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

proximity→get_lowestValue() YProximity
proximity→lowestValue()proximity.get_lowestValue()

Returns the minimal value observed for the proximity detection since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the proximity detection since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

proximity→get_module() YProximity
proximity→module()proximity.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

proximity→get_proximityReportMode() YProximity
proximity→proximityReportMode()
proximity.get_proximityReportMode()

Returns the parameter (sensor value, presence or pulse count) returned by the get_currentValue function and callbacks.

function **get_proximityReportMode**( )

Returns :

a value among `Y_PROXIMITYREPORTMODE_NUMERIC`, `Y_PROXIMITYREPORTMODE_PRESENCE` and `Y_PROXIMITYREPORTMODE_PULSECOUNT` corresponding to the parameter (sensor value, presence or pulse count) returned by the get_currentValue function and callbacks

On failure, throws an exception or returns `Y_PROXIMITYREPORTMODE_INVALID`.

proximity→get_pulseCounter() YProximity
proximity→pulseCounter()
proximity.get_pulseCounter()

Returns the pulse counter value.

function **get_pulseCounter**( )

The value is a 32 bit integer. In case of overflow (>=2^32), the counter will wrap. To reset the counter, just call the resetCounter() method.

Returns :
an integer corresponding to the pulse counter value

On failure, throws an exception or returns `Y_PULSECOUNTER_INVALID`.

proximity→get_pulseTimer() YProximity
proximity→pulseTimer()proximity.get_pulseTimer()

Returns the timer of the pulse counter (ms).

function **get_pulseTimer**( )

Returns :
an integer corresponding to the timer of the pulse counter (ms)

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

proximity→get_recordedData() YProximity
proximity→recordedData()
proximity.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

proximity→get_reportFrequency() YProximity
proximity→reportFrequency()
proximity.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

proximity→get_resolution() YProximity
proximity→resolution()proximity.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

proximity→get_sensorState() YProximity
proximity→sensorState()proximity.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

proximity→get_signalValue() YProximity
proximity→signalValue()proximity.get_signalValue()

Returns the current value of signal measured by the proximity sensor.

function **get_signalValue**( )

Returns :
a floating point number corresponding to the current value of signal measured by the proximity sensor

On failure, throws an exception or returns Y_SIGNALVALUE_INVALID.

proximity→get_unit() YProximity
proximity→unit()proximity.get_unit()

Returns the measuring unit for the proximity detection.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the proximity detection

On failure, throws an exception or returns `Y_UNIT_INVALID`.

proximity→get_userData() YProximity
proximity→userData()proximity.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.

proximity→isOnline()proximity.isOnline() YProximity

Checks if the proximity sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the proximity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the proximity sensor.

Returns :

`true` if the proximity sensor can be reached, and `false` otherwise

proximity→load()proximity.load() YProximity

Preloads the proximity sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

proximity→loadAttribute()proximity.loadAttribute() YProximity

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

proximity→loadCalibrationPoints() YProximity
proximity.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→muteValueCallbacks() YProximity
proximity.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→nextProximity()proximity.nextProximity() YProximity

Continues the enumeration of proximity sensors started using `yFirstProximity()`.

function **nextProximity**( )

Returns :

a pointer to a `YProximity` object, corresponding to a proximity sensor currently online, or a `null` pointer if there are no more proximity sensors to enumerate.

proximity→registerTimedReportCallback() YProximity
proximity.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

proximity→registerValueCallback() YProximity
proximity.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.

proximity→resetCounter()proximity.resetCounter() YProximity

Resets the pulse counter value as well as its timer.

function **resetCounter**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_detectionThreshold() YProximity
proximity→setDetectionThreshold()
proximity.set_detectionThreshold()

Changes the threshold used to determine the logical state of the proximity sensor, when considered as a binary input (on/off).

function **set_detectionThreshold**(**newval**)

Parameters :

newval an integer corresponding to the threshold used to determine the logical state of the proximity sensor, when considered as a binary input (on/off)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_highestValue() YProximity
proximity→setHighestValue()
proximity.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.

proximity→set_logFrequency() YProximity
proximity→setLogFrequency()
proximity.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_logicalName() YProximity
proximity→setLogicalName()
proximity.set_logicalName()

Changes the logical name of the proximity 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 proximity sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_lowestValue() YProximity
proximity→setLowestValue()
proximity.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.

proximity→set_proximityReportMode() YProximity
proximity→setProximityReportMode()
proximity.set_proximityReportMode()

Modifies the parameter type (sensor value, presence or pulse count) returned by the get_currentValue function and callbacks.

function **set_proximityReportMode**(**newval**)

The edge count value is limited to the 6 lowest digits. For values greater than one million, use get_pulseCounter().

Parameters :

newval a value among Y_PROXIMITYREPORTMODE_NUMERIC, Y_PROXIMITYREPORTMODE_PRESENCE and Y_PROXIMITYREPORTMODE_PULSECOUNT

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_reportFrequency() YProximity
proximity→setReportFrequency()
proximity.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_resolution() YProximity
proximity→setResolution()proximity.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→set_userData() YProximity
proximity→setUserData()proximity.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

proximity→startDataLogger() YProximity
proximity.startDataLogger()

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

proximity→stopDataLogger() YProximity
proximity.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

proximity→unmuteValueCallbacks() YProximity
proximity.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

proximity→wait_async()proximity.wait_async() YProximity

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.49. PwmInput function interface

The Yoctopuce class YPwmInput allows you to read and configure Yoctopuce PWM sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to configure the signal parameter used to transmit information: the duty cycle, the frequency or the pulse width.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwminput.js'></script>
cpp	#include "yocto_pwminput.h"
m	#import "yocto_pwminput.h"
pas	uses yocto_pwminput;
vb	yocto_pwminput.vb
cs	yocto_pwminput.cs
java	import com.yoctopuce.YoctoAPI.YPwmInput;
uwp	import com.yoctopuce.YoctoAPI.YPwmInput;
py	from yocto_pwminput import *
php	require_once('yocto_pwminput.php');
es	in HTML: <script src="../../lib/yocto_pwminput.js"></script> in node.js: require('yoctolib-es2017/yocto_pwminput.js');

Global functions
yFindPwmInput(**func**) Retrieves a PWM input for a given identifier.
yFindPwmInputInContext(**yctx**, **func**) Retrieves a PWM input for a given identifier in a YAPI context.
yFirstPwmInput() Starts the enumeration of PWM inputs currently accessible.
yFirstPwmInputInContext(**yctx**) Starts the enumeration of PWM inputs currently accessible.
`YPwmInput` methods
pwminput→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
pwminput→clearCache() Invalidates the cache.
pwminput→describe() Returns a short text that describes unambiguously the instance of the PWM input in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pwminput→get_advertisedValue() Returns the current value of the PWM input (no more than 6 characters).
pwminput→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in HZ, as a floating point number.
pwminput→get_currentValue() Returns the current value of the PwmInput feature as a floating point number.
pwminput→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
pwminput→get_dutyCycle() Returns the PWM duty cycle, in per cents.

pwminput→get_errorMessage()

Returns the error message of the latest error with the PWM input.

pwminput→get_errorType()

Returns the numerical error code of the latest error with the PWM input.

pwminput→get_frequency()

Returns the PWM frequency in Hz.

pwminput→get_friendlyName()

Returns a global identifier of the PWM input in the format `MODULE_NAME.FUNCTION_NAME`.

pwminput→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pwminput→get_functionId()

Returns the hardware identifier of the PWM input, without reference to the module.

pwminput→get_hardwareId()

Returns the unique hardware identifier of the PWM input in the form `SERIAL.FUNCTIONID`.

pwminput→get_highestValue()

Returns the maximal value observed for the PWM since the device was started.

pwminput→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pwminput→get_logicalName()

Returns the logical name of the PWM input.

pwminput→get_lowestValue()

Returns the minimal value observed for the PWM since the device was started.

pwminput→get_module()

Gets the `YModule` object for the device on which the function is located.

pwminput→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwminput→get_period()

Returns the PWM period in milliseconds.

pwminput→get_pulseCounter()

Returns the pulse counter value.

pwminput→get_pulseDuration()

Returns the PWM pulse length in milliseconds, as a floating point number.

pwminput→get_pulseTimer()

Returns the timer of the pulses counter (ms).

pwminput→get_pwmReportMode()

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.

pwminput→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pwminput→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pwminput→get_resolution()

Returns the resolution of the measured values.

pwminput→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

pwminput→get_unit()

Returns the measuring unit for the values returned by get_currentValue and callbacks.

pwminput→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwminput→isOnline()

Checks if the PWM input is currently reachable, without raising any error.

pwminput→isOnline_async(**callback**, **context**)

Checks if the PWM input is currently reachable, without raising any error (asynchronous version).

pwminput→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

pwminput→load(**msValidity**)

Preloads the PWM input cache with a specified validity duration.

pwminput→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

pwminput→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

pwminput→load_async(**msValidity**, **callback**, **context**)

Preloads the PWM input cache with a specified validity duration (asynchronous version).

pwminput→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

pwminput→nextPwmInput()

Continues the enumeration of PWM inputs started using `yFirstPwmInput()`.

pwminput→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

pwminput→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

pwminput→resetCounter()

Returns the pulse counter value as well as its timer.

pwminput→set_highestValue(**newval**)

Changes the recorded maximal value observed.

pwminput→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

pwminput→set_logicalName(**newval**)

Changes the logical name of the PWM input.

pwminput→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

pwminput→set_pwmReportMode(**newval**)

Modifies the parameter type (frequency/duty cycle, pulse width, or edge count) returned by the get_currentValue function and callbacks.

pwminput→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

pwminput→set_resolution(**newval**)

Changes the resolution of the measured physical values.

pwminput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwminput→startDataLogger()

Starts the data logger on the device.

pwminput→stopDataLogger()

Stops the datalogger on the device.

pwminput→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

pwminput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmInput.FindPwmInput() YPwmInput yFindPwmInput()yFindPwmInput()

Retrieves a PWM input for a given identifier.

function **FindPwmInput**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmInput.isOnline()` to test if the PWM input is indeed online at a given time. In case of ambiguity when looking for a PWM 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the PWM input

Returns :

a `YPwmInput` object allowing you to drive the PWM input.

YPwmInput.FindPwmInputInContext() YPwmInput yFindPwmInputInContext()

Retrieves a PWM input for a given identifier in a YAPI context.

function **FindPwmInputInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmInput.isOnline()` to test if the PWM input is indeed online at a given time. In case of ambiguity when looking for a PWM 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 :

yctx a YAPI context

func a string that uniquely characterizes the PWM input

Returns :

a `YPwmInput` object allowing you to drive the PWM input.

YPwmInput.FirstPwmInput() YPwmInput yFirstPwmInput()yFirstPwmInput()

Starts the enumeration of PWM inputs currently accessible.

function **FirstPwmInput**( )

Use the method `YPwmInput.nextPwmInput()` to iterate on next PWM inputs.

Returns :

a pointer to a `YPwmInput` object, corresponding to the first PWM input currently online, or a `null` pointer if there are none.

YPwmInput.FirstPwmInputInContext() YPwmInput yFirstPwmInputInContext()

Starts the enumeration of PWM inputs currently accessible.

function **FirstPwmInputInContext**(**yctx**)

Use the method YPwmInput.nextPwmInput() to iterate on next PWM inputs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a YPwmInput object, corresponding to the first PWM input currently online, or a null pointer if there are none.

pwminput→calibrateFromPoints() YPwmInput
pwminput.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→clearCache()pwminput.clearCache() YPwmInput

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the PWM input attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

pwminput→describe()pwminput.describe() YPwmInput

Returns a short text that describes unambiguously the instance of the PWM input in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the PWM input (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwminput→get_advertisedValue() YPwmInput
pwminput→advertisedValue()
pwminput.get_advertisedValue()

Returns the current value of the PWM input (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the PWM input (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwminput→get_currentRawValue() YPwmInput
pwminput→currentRawValue()
pwminput.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in HZ, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in HZ, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

pwminput→get_currentValue() YPwmInput
pwminput→currentValue()
pwminput.get_currentValue()

Returns the current value of the PwmInput feature as a floating point number.

function **get_currentValue**( )

Depending on the pwmReportMode setting, this can be the frequency, in Hz, the duty cycle in % or the pulse length in ms.

Returns :
a floating point number corresponding to the current value of the PwmInput feature as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

pwminput→get_dataLogger() YPwmInput
pwminput→dataLogger()pwminput.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

pwminput→get_dutyCycle() YPwmInput
pwminput→dutyCycle()pwminput.get_dutyCycle()

Returns the PWM duty cycle, in per cents.

function **get_dutyCycle**( )

Returns :
a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns `Y_DUTYCYCLE_INVALID`.

pwminput→get_errorMessage() YPwmInput
pwminput→errorMessage()
pwminput.get_errorMessage()

Returns the error message of the latest error with the PWM input.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the PWM input object

pwminput→get_errorType() YPwmInput
pwminput→errorType()pwminput.get_errorType()

Returns the numerical error code of the latest error with the PWM input.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the PWM input object

pwminput→get_frequency() YPwmInput
pwminput→frequency()pwminput.get_frequency()

Returns the PWM frequency in Hz.

function **get_frequency**( )

Returns :
a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`.

pwminput→get_friendlyName() YPwmInput
pwminput→friendlyName()
pwminput.get_friendlyName()

Returns a global identifier of the PWM input in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the PWM input if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM input (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the PWM input using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwminput→get_functionDescriptor() YPwmInput
pwminput→functionDescriptor()
pwminput.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

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.

pwminput→get_functionId() YPwmInput
pwminput→functionId()pwminput.get_functionId()

Returns the hardware identifier of the PWM input, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the PWM input (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwminput→get_hardwareId() YPwmInput
pwminput→hardwareId()pwminput.get_hardwareId()

Returns the unique hardware identifier of the PWM input in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM input (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the PWM input (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pwminput→get_highestValue() YPwmInput
pwminput→highestValue()
pwminput.get_highestValue()

Returns the maximal value observed for the PWM since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the PWM since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

pwminput→get_logFrequency() YPwmInput
pwminput→logFrequency()
pwminput.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

pwminput→get_logicalName() YPwmInput
pwminput→logicalName()
pwminput.get_logicalName()

Returns the logical name of the PWM input.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the PWM input.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

pwminput→get_lowestValue() YPwmInput
pwminput→lowestValue()
pwminput.get_lowestValue()

Returns the minimal value observed for the PWM since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the PWM since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

pwminput→get_module() YPwmInput
pwminput→module()pwminput.get_module()

Gets the YModule object for the device on which the function is located.

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

pwminput→get_period() YPwmInput
pwminput→period()pwminput.get_period()

Returns the PWM period in milliseconds.

function **get_period**( )

Returns :
a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

pwminput→get_pulseCounter()
pwminput→pulseCounter()
pwminput.get_pulseCounter()

YPwmInput

Returns the pulse counter value.

function **get_pulseCounter**( )

Actually that counter is incremented twice per period. That counter is limited to 1 billion

Returns :
an integer corresponding to the pulse counter value

On failure, throws an exception or returns `Y_PULSECOUNTER_INVALID`.

pwminput→get_pulseDuration() YPwmInput
pwminput→pulseDuration()
pwminput.get_pulseDuration()

Returns the PWM pulse length in milliseconds, as a floating point number.

function **get_pulseDuration**( )

Returns :
a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`.

pwminput→get_pulseTimer() YPwmInput
pwminput→pulseTimer()pwminput.get_pulseTimer()

Returns the timer of the pulses counter (ms).

function **get_pulseTimer**( )

Returns :
an integer corresponding to the timer of the pulses counter (ms)

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

pwminput→get_pwmReportMode() YPwmInput
pwminput→pwmReportMode()
pwminput.get_pwmReportMode()

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.

function **get_pwmReportMode**( )

Attention

Returns :

a value among `Y_PWMREPORTMODE_PWM_DUTYCYCLE`, `Y_PWMREPORTMODE_PWM_FREQUENCY`, `Y_PWMREPORTMODE_PWM_PULSEDURATION` and `Y_PWMREPORTMODE_PWM_EDGECOUNT` corresponding to the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks

On failure, throws an exception or returns `Y_PWMREPORTMODE_INVALID`.

pwminput→get_recordedData() YPwmInput
pwminput→recordedData()
pwminput.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pwminput→get_reportFrequency() YPwmInput
pwminput→reportFrequency()
pwminput.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

pwminput→get_resolution() YPwmInput
pwminput→resolution()pwminput.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

pwminput→get_sensorState() YPwmInput
pwminput→sensorState()
pwminput.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

pwminput→get_unit() YPwmInput
pwminput→unit()pwminput.get_unit()

Returns the measuring unit for the values returned by get_currentValue and callbacks.

function **get_unit**( )

This unit changes according to the pwmReportMode settings.

Returns :
a string corresponding to the measuring unit for the values returned by get_currentValue and callbacks

On failure, throws an exception or returns `Y_UNIT_INVALID`.

pwminput→get_userData() YPwmInput
pwminput→userData()pwminput.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

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.

pwminput→isOnline()pwminput.isOnline() YPwmInput

Checks if the PWM input is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the PWM input in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM input.

Returns :
`true` if the PWM input can be reached, and `false` otherwise

pwminput→load()pwminput.load() YPwmInput

Preloads the PWM input cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→loadAttribute()pwminput.loadAttribute() YPwmInput

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

pwminput→loadCalibrationPoints() YPwmInput
pwminput.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→muteValueCallbacks() YPwmInput
pwminput.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→nextPwmInput() YPwmInput
pwminput.nextPwmInput()

Continues the enumeration of PWM inputs started using `yFirstPwmInput()`.

function **nextPwmInput**( )

Returns :

a pointer to a `YPwmInput` object, corresponding to a PWM input currently online, or a `null` pointer if there are no more PWM inputs to enumerate.

pwminput→registerTimedReportCallback() YPwmInput
pwminput.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

pwminput→registerValueCallback() YPwmInput
pwminput.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.

pwminput→resetCounter()pwminput.resetCounter() YPwmInput

Returns the pulse counter value as well as its timer.

function **resetCounter**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_highestValue() YPwmInput
pwminput→setHighestValue()
pwminput.set_highestValue()

Changes the recorded maximal value observed.

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.

pwminput→set_logFrequency() YPwmInput
pwminput→setLogFrequency()
pwminput.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_logicalName() YPwmInput
pwminput→setLogicalName()
pwminput.set_logicalName()

Changes the logical name of the PWM 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 PWM input.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_lowestValue()
pwminput→setLowestValue()
pwminput.set_lowestValue()

YPwmInput

Changes the recorded minimal value observed.

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.

pwminput→set_pwmReportMode() YPwmInput
pwminput→setPwmReportMode()
pwminput.set_pwmReportMode()

Modifies the parameter type (frequency/duty cycle, pulse width, or edge count) returned by the get_currentValue function and callbacks.

function **set_pwmReportMode**(**newval**)

The edge count value is limited to the 6 lowest digits. For values greater than one million, use get_pulseCounter().

Parameters :

newval a value among Y_PWMREPORTMODE_PWM_DUTYCYCLE, Y_PWMREPORTMODE_PWM_FREQUENCY, Y_PWMREPORTMODE_PWM_PULSEDURATION and Y_PWMREPORTMODE_PWM_EDGECOUNT

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_reportFrequency() YPwmInput
pwminput→setReportFrequency()
pwminput.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_resolution() YPwmInput
pwminput→setResolution()
pwminput.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_userData() YPwmInput
pwminput→setUserData()pwminput.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

pwminput→startDataLogger() YPwmInput
pwminput.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

pwminput→stopDataLogger() YPwmInput
pwminput.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

pwminput→unmuteValueCallbacks() YPwmInput
pwminput.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→wait_async()pwminput.wait_async() YPwmInput

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.50. PwmOutput function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmoutput.js'></script>
cpp	#include "yocto_pwmoutput.h"
m	#import "yocto_pwmoutput.h"
pas	uses yocto_pwmoutput;
vb	yocto_pwmoutput.vb
cs	yocto_pwmoutput.cs
java	import com.yoctopuce.YoctoAPI.YPwmOutput;
uwp	import com.yoctopuce.YoctoAPI.YPwmOutput;
py	from yocto_pwmoutput import *
php	require_once('yocto_pwmoutput.php');
es	in HTML: <script src="../../lib/yocto_pwmoutput.js"></script> in node.js: require('yoctolib-es2017/yocto_pwmoutput.js');

Global functions

yFindPwmOutput(**func**)

Retrieves a PWM for a given identifier.

yFindPwmOutputInContext(**yctx**, **func**)

Retrieves a PWM for a given identifier in a YAPI context.

yFirstPwmOutput()

Starts the enumeration of PWMs currently accessible.

yFirstPwmOutputInContext(**yctx**)

Starts the enumeration of PWMs currently accessible.

YPwmOutput methods

pwmoutput→clearCache()

Invalidates the cache.

pwmoutput→describe()

Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

pwmoutput→dutyCycleMove(**target**, **ms_duration**)

Performs a smooth change of the pulse duration toward a given value.

pwmoutput→get_advertisedValue()

Returns the current value of the PWM (no more than 6 characters).

pwmoutput→get_dutyCycle()

Returns the PWM duty cycle, in per cents.

pwmoutput→get_dutyCycleAtPowerOn()

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100.

pwmoutput→get_enabled()

Returns the state of the PWMs.

pwmoutput→get_enabledAtPowerOn()

Returns the state of the PWM at device power on.

pwmoutput→get_errorMessage()

Returns the error message of the latest error with the PWM.

pwmoutput→get_errorType()

Returns the numerical error code of the latest error with the PWM.

pwmoutput→get_frequency()

Returns the PWM frequency in Hz.

pwmoutput→get_friendlyName()

Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.

pwmoutput→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pwmoutput→get_functionId()

Returns the hardware identifier of the PWM, without reference to the module.

pwmoutput→get_hardwareId()

Returns the unique hardware identifier of the PWM in the form `SERIAL.FUNCTIONID`.

pwmoutput→get_logicalName()

Returns the logical name of the PWM.

pwmoutput→get_module()

Gets the `YModule` object for the device on which the function is located.

pwmoutput→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwmoutput→get_period()

Returns the PWM period in milliseconds.

pwmoutput→get_pulseDuration()

Returns the PWM pulse length in milliseconds, as a floating point number.

pwmoutput→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwmoutput→isOnline()

Checks if the PWM is currently reachable, without raising any error.

pwmoutput→isOnline_async(**callback**, **context**)

Checks if the PWM is currently reachable, without raising any error (asynchronous version).

pwmoutput→load(**msValidity**)

Preloads the PWM cache with a specified validity duration.

pwmoutput→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

pwmoutput→load_async(**msValidity**, **callback**, **context**)

Preloads the PWM cache with a specified validity duration (asynchronous version).

pwmoutput→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

pwmoutput→nextPwmOutput()

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

pwmoutput→pulseDurationMove(**ms_target**, **ms_duration**)

Performs a smooth transistion of the pulse duration toward a given value.

pwmoutput→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

pwmoutput→set_dutyCycle(**newval**)

Changes the PWM duty cycle, in per cents.

pwmoutput→set_dutyCycleAtPowerOn(**newval**)

Changes the PWM duty cycle at device power on.

pwmoutput→set_enabled(newval)

Stops or starts the PWM.

pwmoutput→set_enabledAtPowerOn(newval)

Changes the state of the PWM at device power on.

pwmoutput→set_frequency(newval)

Changes the PWM frequency.

pwmoutput→set_logicalName(newval)

Changes the logical name of the PWM.

pwmoutput→set_period(newval)

Changes the PWM period in milliseconds.

pwmoutput→set_pulseDuration(newval)

Changes the PWM pulse length, in milliseconds.

pwmoutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmoutput→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

pwmoutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmOutput.FindPwmOutput() YPwmOutput yFindPwmOutput()yFindPwmOutput()

Retrieves a PWM for a given identifier.

function **FindPwmOutput**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the PWM

Returns :

a `YPwmOutput` object allowing you to drive the PWM.

YPwmOutput.FindPwmOutputInContext() YPwmOutput yFindPwmOutputInContext()

Retrieves a PWM for a given identifier in a YAPI context.

function **FindPwmOutputInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the PWM

Returns :

a `YPwmOutput` object allowing you to drive the PWM.

YPwmOutput.FirstPwmOutput() YPwmOutput yFirstPwmOutput()yFirstPwmOutput()

Starts the enumeration of PWMs currently accessible.

function **FirstPwmOutput**( )

Use the method YPwmOutput.nextPwmOutput() to iterate on next PWMs.

Returns :

a pointer to a YPwmOutput object, corresponding to the first PWM currently online, or a null pointer if there are none.

YPwmOutput.FirstPwmOutputInContext() YPwmOutput yFirstPwmOutputInContext()

Starts the enumeration of PWMs currently accessible.

function **FirstPwmOutputInContext**(**yctx**)

Use the method `YPwmOutput.nextPwmOutput()` to iterate on next PWMs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YPwmOutput` object, corresponding to the first PWM currently online, or a `null` pointer if there are none.

pwmoutput→clearCache()pwmoutput.clearCache() YPwmOutput

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the PWM attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

pwmoutput→describe()pwmoutput.describe() YPwmOutput

Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the PWM (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmoutput→dutyCycleMove() YPwmOutput
pwmoutput.dutyCycleMove()

Performs a smooth change of the pulse duration toward a given value.

function **dutyCycleMove**(**target**, **ms_duration**)

Parameters :

target	new duty cycle at the end of the transition (floating-point number, between 0 and 1)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→get_advertisedValue() YPwmOutput
pwmoutput→advertisedValue()
pwmoutput.get_advertisedValue()

Returns the current value of the PWM (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the PWM (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

pwmoutput→get_dutyCycle() YPwmOutput
pwmoutput→dutyCycle()pwmoutput.get_dutyCycle()

Returns the PWM duty cycle, in per cents.

function **get_dutyCycle**( )

Returns :
a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns `Y_DUTYCYCLE_INVALID`.

pwmoutput→get_dutyCycleAtPowerOn() YPwmOutput
pwmoutput→dutyCycleAtPowerOn()
pwmoutput.get_dutyCycleAtPowerOn()

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100.

function **get_dutyCycleAtPowerOn**( )

Returns :

a floating point number corresponding to the PWMs duty cycle at device power on as a floating point number between 0 and 100

On failure, throws an exception or returns `Y_DUTYCYCLEATPOWERON_INVALID`.

pwmoutput→get_enabled() YPwmOutput
pwmoutput→enabled()pwmoutput.get_enabled()

Returns the state of the PWMs.

function **get_enabled**( )

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the PWMs

On failure, throws an exception or returns Y_ENABLED_INVALID.

pwmoutput→get_enabledAtPowerOn() YPwmOutput
pwmoutput→enabledAtPowerOn()
pwmoutput.get_enabledAtPowerOn()

Returns the state of the PWM at device power on.

function **get_enabledAtPowerOn**( )

Returns :

either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the state of the PWM at device power on

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.

pwmoutput→get_errorMessage() YPwmOutput
pwmoutput→errorMessage()
pwmoutput.get_errorMessage()

Returns the error message of the latest error with the PWM.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the PWM object

pwmoutput→get_errorType() YPwmOutput
pwmoutput→errorType()pwmoutput.get_errorType()

Returns the numerical error code of the latest error with the PWM.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the PWM object

pwmoutput→get_frequency() YPwmOutput
pwmoutput→frequency()pwmoutput.get_frequency()

Returns the PWM frequency in Hz.

function **get_frequency**( )

Returns :
a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns Y_FREQUENCY_INVALID.

pwmoutput→get_friendlyName() YPwmOutput
pwmoutput→friendlyName()
pwmoutput.get_friendlyName()

Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the PWM if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the PWM using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwmoutput→get_functionDescriptor() YPwmOutput
pwmoutput→functionDescriptor()
pwmoutput.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmoutput→get_functionId() YPwmOutput
pwmoutput→functionId()pwmoutput.get_functionId()

Returns the hardware identifier of the PWM, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the PWM (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwmoutput→get_hardwareId() YPwmOutput
pwmoutput→hardwareId()
pwmoutput.get_hardwareId()

Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the PWM (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pwmoutput→get_logicalName() YPwmOutput
pwmoutput→logicalName()
pwmoutput.get_logicalName()

Returns the logical name of the PWM.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the PWM.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

pwmoutput→get_module() YPwmOutput
pwmoutput→module()pwmoutput.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

pwmoutput→get_period() YPwmOutput
pwmoutput→period()pwmoutput.get_period()

Returns the PWM period in milliseconds.

function **get_period**( )

Returns :
a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

pwmoutput→get_pulseDuration() YPwmOutput
pwmoutput→pulseDuration()
pwmoutput.get_pulseDuration()

Returns the PWM pulse length in milliseconds, as a floating point number.

function **get_pulseDuration**( )

Returns :
a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`.

pwmoutput→get_userData() YPwmOutput
pwmoutput→userData()pwmoutput.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

pwmoutput→isOnline()pwmoutput.isOnline() YPwmOutput

Checks if the PWM is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

Returns :

`true` if the PWM can be reached, and `false` otherwise

pwmoutput→load()pwmoutput.load() YPwmOutput

Preloads the PWM cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→loadAttribute() YPwmOutput
pwmoutput.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

pwmoutput→muteValueCallbacks() YPwmOutput
pwmoutput.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→nextPwmOutput() YPwmOutput
pwmoutput.nextPwmOutput()

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

function **nextPwmOutput**( )

Returns :

a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.

pwmoutput→pulseDurationMove() YPwmOutput
pwmoutput.pulseDurationMove()

Performs a smooth transistion of the pulse duration toward a given value.

function **pulseDurationMove**(**ms_target**, **ms_duration**)

Any period, frequency, duty cycle or pulse width change will cancel any ongoing transition process.

Parameters :

ms_target	new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→registerValueCallback() YPwmOutput
pwmoutput.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.

pwmoutput→set_dutyCycle() YPwmOutput
pwmoutput→setDutyCycle()
pwmoutput.set_dutyCycle()

Changes the PWM duty cycle, in per cents.

function **set_dutyCycle**(**newval**)

Parameters :

newval a floating point number corresponding to the PWM duty cycle, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_dutyCycleAtPowerOn() YPwmOutput
pwmoutput→setDutyCycleAtPowerOn()
pwmoutput.set_dutyCycleAtPowerOn()

Changes the PWM duty cycle at device power on.

function **set_dutyCycleAtPowerOn**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a floating point number corresponding to the PWM duty cycle at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabled() YPwmOutput
pwmoutput→setEnabled()pwmoutput.set_enabled()

Stops or starts the PWM.

function **set_enabled**(**newval**)

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabledAtPowerOn() YPwmOutput
pwmoutput→setEnabledAtPowerOn()
pwmoutput.set_enabledAtPowerOn()

Changes the state of the PWM at device power on.

function **set_enabledAtPowerOn**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_frequency() YPwmOutput
pwmoutput→setFrequency()
pwmoutput.set_frequency()

Changes the PWM frequency.

function **set_frequency**(**newval**)

The duty cycle is kept unchanged thanks to an automatic pulse width change.

Parameters :

newval a floating point number corresponding to the PWM frequency

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_logicalName() YPwmOutput
pwmoutput→setLogicalName()
pwmoutput.set_logicalName()

Changes the logical name of the PWM.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the PWM.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_period() YPwmOutput
pwmoutput→setPeriod()pwmoutput.set_period()

Changes the PWM period in milliseconds.

function **set_period**(**newval**)

Parameters :

newval a floating point number corresponding to the PWM period in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_pulseDuration() YPwmOutput
pwmoutput→setPulseDuration()
pwmoutput.set_pulseDuration()

Changes the PWM pulse length, in milliseconds.

function **set_pulseDuration**(**newval**)

A pulse length cannot be longer than period, otherwise it is truncated.

Parameters :

newval a floating point number corresponding to the PWM pulse length, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_userData() YPwmOutput
pwmoutput→setUserData()
pwmoutput.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

pwmoutput→unmuteValueCallbacks() YPwmOutput
pwmoutput.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→wait_async()pwmoutput.wait_async() YPwmOutput

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.51. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmpowersource.js'></script>
cpp	#include "yocto_pwmpowersource.h"
m	#import "yocto_pwmpowersource.h"
pas	uses yocto_pwmpowersource;
vb	yocto_pwmpowersource.vb
cs	yocto_pwmpowersource.cs
java	import com.yoctopuce.YoctoAPI.YPwmPowerSource;
uwp	import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py	from yocto_pwmpowersource import *
php	require_once('yocto_pwmpowersource.php');
es	in HTML: <script src="../../lib/yocto_pwmpowersource.js"></script> in node.js: require('yoctolib-es2017/yocto_pwmpowersource.js');

Global functions

yFindPwmPowerSource(**func**)

Retrieves a voltage source for a given identifier.

yFindPwmPowerSourceInContext(**yctx**, **func**)

Retrieves a voltage source for a given identifier in a YAPI context.

yFirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

yFirstPwmPowerSourceInContext(**yctx**)

Starts the enumeration of Voltage sources currently accessible.

YPwmPowerSource methods

pwmpowersource→clearCache()

Invalidates the cache.

pwmpowersource→describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

pwmpowersource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

pwmpowersource→get_errorMessage()

Returns the error message of the latest error with the voltage source.

pwmpowersource→get_errorType()

Returns the numerical error code of the latest error with the voltage source.

pwmpowersource→get_friendlyName()

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

pwmpowersource→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pwmpowersource→get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

pwmpowersource→get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

pwmpowersource→get_logicalName()

Returns the logical name of the voltage source.

pwmpowersource→get_module()

Gets the `YModule` object for the device on which the function is located.

pwmpowersource→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwmpowersource→get_powerMode()

Returns the selected power source for the PWM on the same device.

pwmpowersource→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwmpowersource→isOnline()

Checks if the voltage source is currently reachable, without raising any error.

pwmpowersource→isOnline_async(callback, context)

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

pwmpowersource→load(msValidity)

Preloads the voltage source cache with a specified validity duration.

pwmpowersource→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

pwmpowersource→load_async(msValidity, callback, context)

Preloads the voltage source cache with a specified validity duration (asynchronous version).

pwmpowersource→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

pwmpowersource→nextPwmPowerSource()

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

pwmpowersource→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pwmpowersource→set_logicalName(newval)

Changes the logical name of the voltage source.

pwmpowersource→set_powerMode(newval)

Changes the PWM power source.

pwmpowersource→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmpowersource→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

pwmpowersource→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmPowerSource.FindPwmPowerSource() YPwmPowerSource yFindPwmPowerSource()yFindPwmPowerSource()

Retrieves a voltage source for a given identifier.

```
function FindPwmPowerSource( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmPowerSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the voltage source

Returns :

a `YPwmPowerSource` object allowing you to drive the voltage source.

YPwmPowerSource.FindPwmPowerSourceInContext() YPwmPowerSource yFindPwmPowerSourceInContext()

Retrieves a voltage source for a given identifier in a YAPI context.

function **FindPwmPowerSourceInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmPowerSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the voltage source

Returns :

a `YPwmPowerSource` object allowing you to drive the voltage source.

YPwmPowerSource.FirstPwmPowerSource() YPwmPowerSource yFirstPwmPowerSource()yFirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

function **FirstPwmPowerSource**( )

Use the method `YPwmPowerSource.nextPwmPowerSource()` to iterate on next Voltage sources.

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to the first source currently online, or a `null` pointer if there are none.

YPwmPowerSource.FirstPwmPowerSourceInContext() YPwmPowerSource yFirstPwmPowerSourceInContext()

Starts the enumeration of Voltage sources currently accessible.

function **FirstPwmPowerSourceInContext**(**yctx**)

Use the method `YPwmPowerSource.nextPwmPowerSource()` to iterate on next Voltage sources.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to the first source currently online, or a `null` pointer if there are none.

pwmpowersource→clearCache() YPwmPowerSource
pwmpowersource.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the voltage source attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

pwmpowersource→describe() YPwmPowerSource
pwmpowersource.describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage source (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmpowersource→get_advertisedValue() YPwmPowerSource
pwmpowersource→advertisedValue()
pwmpowersource.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`.

pwmpowersource→get_errorMessage() YPwmPowerSource
pwmpowersource→errorMessage()
pwmpowersource.get_errorMessage()

Returns the error message of the latest error with the voltage source.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the voltage source object

pwmpowersource→get_errorType() YPwmPowerSource
pwmpowersource→errorType()
pwmpowersource.get_errorType()

Returns the numerical error code of the latest error with the voltage source.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the voltage source object

pwmpowersource→get_friendlyName() YPwmPowerSource
pwmpowersource→friendlyName()
pwmpowersource.get_friendlyName()

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the voltage source if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage source (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage source using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwmpowersource→get_functionDescriptor() YPwmPowerSource
pwmpowersource→functionDescriptor()
pwmpowersource.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmpowersource→get_functionId() YPwmPowerSource
pwmpowersource→functionId()
pwmpowersource.get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the voltage source (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwmpowersource→get_hardwareId() YPwmPowerSource
pwmpowersource→hardwareId()
pwmpowersource.get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage source (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the voltage source (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pwmpowersource→get_logicalName() YPwmPowerSource
pwmpowersource→logicalName()
pwmpowersource.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`.

pwmpowersource→get_module()
pwmpowersource→module()
pwmpowersource.get_module()

YPwmPowerSource

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

pwmpowersource→get_powerMode() YPwmPowerSource
pwmpowersource→powerMode()
pwmpowersource.get_powerMode()

Returns the selected power source for the PWM on the same device.

function **get_powerMode**( )

Returns :

a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns `Y_POWERMODE_INVALID`.

pwmpowersource→get_userData() YPwmPowerSource
pwmpowersource→userData()
pwmpowersource.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.

pwmpowersource→isOnline() YPwmPowerSource
pwmpowersource.isOnline()

Checks if the voltage source is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

Returns :

`true` if the voltage source can be reached, and `false` otherwise

pwmpowersource→load()pwmpowersource.load() YPwmPowerSource

Preloads the voltage source cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→loadAttribute() YPwmPowerSource
pwmpowersource.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

pwmpowersource→muteValueCallbacks() YPwmPowerSource
pwmpowersource.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→nextPwmPowerSource() YPwmPowerSource
pwmpowersource.nextPwmPowerSource()

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

function **nextPwmPowerSource**( )

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more Voltage sources to enumerate.

pwmpowersource→registerValueCallback() YPwmPowerSource
pwmpowersource.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.

pwmpowersource→set_logicalName() YPwmPowerSource
pwmpowersource→setLogicalName()
pwmpowersource.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.

pwmpowersource→set_powerMode() YPwmPowerSource
pwmpowersource→setPowerMode()
pwmpowersource.set_powerMode()

Changes the PWM power source.

function **set_powerMode**(**newval**)

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

Parameters :

newval a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the PWM power source

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→set_userData() YPwmPowerSource
pwmpowersource→setUserData()
pwmpowersource.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

pwmpowersource→unmuteValueCallbacks() YPwmPowerSource
pwmpowersource.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→wait_async() YPwmPowerSource
pwmpowersource.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.52. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>
cpp	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
uwp	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *
php	require_once('yocto_gyro.php');
es	in HTML: <script src="../../lib/yocto_gyro.js"></script> in node.js: require('yoctolib-es2017/yocto_gyro.js');

Global functions

yFindQt(**func**)

Retrieves a quaternion component for a given identifier.

yFindQtInContext(**yctx**, **func**)

Retrieves a quaternion component for a given identifier in a YAPI context.

yFirstQt()

Starts the enumeration of quaternion components currently accessible.

yFirstQtInContext(**yctx**)

Starts the enumeration of quaternion components currently accessible.

YQt methods

qt→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

qt→clearCache()

Invalidates the cache.

qt→describe()

Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

qt→get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

qt→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.

qt→get_currentValue()

Returns the current value of the value, in units, as a floating point number.

qt→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

qt→get_errorMessage()

Returns the error message of the latest error with the quaternion component.

qt→get_errorType()

Returns the numerical error code of the latest error with the quaternion component.

qt→get_friendlyName()

Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.

qt→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

qt→get_functionId()

Returns the hardware identifier of the quaternion component, without reference to the module.

qt→get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

qt→get_highestValue()

Returns the maximal value observed for the value since the device was started.

qt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

qt→get_logicalName()

Returns the logical name of the quaternion component.

qt→get_lowestValue()

Returns the minimal value observed for the value since the device was started.

qt→get_module()

Gets the `YModule` object for the device on which the function is located.

qt→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

qt→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

qt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

qt→get_resolution()

Returns the resolution of the measured values.

qt→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

qt→get_unit()

Returns the measuring unit for the value.

qt→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

qt→isOnline()

Checks if the quaternion component is currently reachable, without raising any error.

qt→isOnline_async(**callback**, **context**)

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

qt→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

qt→load(**msValidity**)

Preloads the quaternion component cache with a specified validity duration.

qt→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

qt→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

qt→load_async(**msValidity**, **callback**, **context**)

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

qt→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

qt→nextQt()

Continues the enumeration of quaternion components started using `yFirstQt()`.

qt→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

qt→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

qt→set_highestValue(**newval**)

Changes the recorded maximal value observed.

qt→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

qt→set_logicalName(**newval**)

Changes the logical name of the quaternion component.

qt→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

qt→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

qt→set_resolution(**newval**)

Changes the resolution of the measured physical values.

qt→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

qt→startDataLogger()

Starts the data logger on the device.

qt→stopDataLogger()

Stops the datalogger on the device.

qt→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

qt→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YQt.FindQt() YQt yFindQt()yFindQt()

Retrieves a quaternion component for a given identifier.

function **FindQt**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQt.isOnline()` to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the quaternion component

Returns :

a `YQt` object allowing you to drive the quaternion component.

YQt.FindQtInContext() YQt
yFindQtInContext()

Retrieves a quaternion component for a given identifier in a YAPI context.

function **FindQtInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQt.isOnline()` to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the quaternion component

Returns :

a `YQt` object allowing you to drive the quaternion component.

YQt.FirstQt() YQt yFirstQt()yFirstQt()

Starts the enumeration of quaternion components currently accessible.

```
function FirstQt( )
```

Use the method `YQt.nextQt()` to iterate on next quaternion components.

Returns :

a pointer to a `YQt` object, corresponding to the first quaternion component currently online, or a `null` pointer if there are none.

YQt.FirstQtInContext() YQt
yFirstQtInContext()

Starts the enumeration of quaternion components currently accessible.

function **FirstQtInContext**(**yctx**)

Use the method `YQt.nextQt()` to iterate on next quaternion components.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YQt` object, corresponding to the first quaternion component currently online, or a `null` pointer if there are none.

qt→calibrateFromPoints()qt.calibrateFromPoints() YQt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

function **calibrateFromPoints**(**rawValues**, **refValues**)

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→clearCache()qt.clearCache() YQt

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the quaternion component attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

qt→describe()qt.describe() YQt

Returns a short text that describes unambiguously the instance of the quaternion component in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the quaternion component (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

qt→get_advertisedValue() YQt
qt→advertisedValue()qt.get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the quaternion component (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

qt→get_currentRawValue() YQt
qt→currentRawValue()qt.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

qt→get_currentValue() YQt
qt→currentValue()qt.get_currentValue()

Returns the current value of the value, in units, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the value, in units, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

qt→get_dataLogger() YQt
qt→dataLogger()qt.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

qt→get_errorMessage() YQt
qt→errorMessage()qt.get_errorMessage()

Returns the error message of the latest error with the quaternion component.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the quaternion component object

qt→get_errorType() YQt
qt→errorType()qt.get_errorType()

Returns the numerical error code of the latest error with the quaternion component.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the quaternion component object

qt→get_friendlyName() YQt
qt→friendlyName()qt.get_friendlyName()

Returns a global identifier of the quaternion component in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the quaternion component if they are defined, otherwise the serial number of the module and the hardware identifier of the quaternion component (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the quaternion component using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

qt→get_functionDescriptor() YQt
qt→functionDescriptor()qt.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

qt→get_functionId() YQt
qt→functionId()qt.get_functionId()

Returns the hardware identifier of the quaternion component, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the quaternion component (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

qt→get_hardwareId() YQt
qt→hardwareId()qt.get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quaternion component (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the quaternion component (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

qt→get_highestValue() YQt
qt→highestValue()qt.get_highestValue()

Returns the maximal value observed for the value since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

qt→get_logFrequency() YQt
qt→logFrequency()qt.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

qt→get_logicalName() YQt
qt→logicalName()qt.get_logicalName()

Returns the logical name of the quaternion component.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the quaternion component.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

qt→get_lowestValue() YQt
qt→lowestValue()qt.get_lowestValue()

Returns the minimal value observed for the value since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the value since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

qt→get_module() YQt
qt→module()qt.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

qt→get_recordedData() YQt
qt→recordedData()qt.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

qt→get_reportFrequency() YQt
qt→reportFrequency()qt.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

qt→get_resolution() YQt
qt→resolution()qt.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

qt→get_sensorState() YQt
qt→sensorState()qt.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

qt→get_unit() YQt
qt→unit()qt.get_unit()

Returns the measuring unit for the value.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the value

On failure, throws an exception or returns `Y_UNIT_INVALID`.

qt→get_userData() YQt
qt→userData()qt.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.

qt→isOnline()qt.isOnline() YQt

Checks if the quaternion component is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

Returns :

`true` if the quaternion component can be reached, and `false` otherwise

qt→isSensorReady() YQt

Checks if the sensor is currently able to provide an up-to-date measure.

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit.
No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

Returns :

`true` if the sensor can provide an up-to-date measure, and `false` otherwise

qt→load()qt.load() YQt

Preloads the quaternion component cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→loadAttribute()qt.loadAttribute() YQt

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

qt→loadCalibrationPoints()qt.loadCalibrationPoints() YQt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→muteValueCallbacks()qt.muteValueCallbacks() YQt

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→nextQt()qt.nextQt() YQt

Continues the enumeration of quaternion components started using `yFirstQt()`.

function **nextQt**( )

Returns :

a pointer to a `YQt` object, corresponding to a quaternion component currently online, or a `null` pointer if there are no more quaternion components to enumerate.

qt→registerTimedReportCallback() YQt
qt.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

qt→registerValueCallback() YQt
qt.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.

qt→set_highestValue() YQt
qt→setHighestValue()qt.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.

qt→set_logFrequency() YQt
qt→setLogFrequency()qt.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logicalName() YQt
qt→setLogicalName()qt.set_logicalName()

Changes the logical name of the quaternion component.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the quaternion component.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_lowestValue() YQt
qt→setLowestValue()qt.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.

qt→set_reportFrequency() YQt
qt→setReportFrequency()qt.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_resolution() YQt
qt→setResolution()qt.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_userData() YQt
qt→setUserData()qt.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

qt→startDataLogger()qt.startDataLogger() YQt

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

qt→stopDataLogger()qt.stopDataLogger() YQt

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

qt→unmuteValueCallbacks() YQt
qt.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→wait_async()qt.wait_async() YQt

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.53. QuadratureDecoder function interface

The class YQuadratureDecoder allows you to decode a two-wire signal produced by a quadrature encoder. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_quadraturedecoder.js'></script>
cpp	#include "yocto_quadraturedecoder.h"
m	#import "yocto_quadraturedecoder.h"
pas	uses yocto_quadraturedecoder;
vb	yocto_quadraturedecoder.vb
cs	yocto_quadraturedecoder.cs
java	import com.yoctopuce.YoctoAPI.YQuadratureDecoder;
uwp	import com.yoctopuce.YoctoAPI.YQuadratureDecoder;
py	from yocto_quadraturedecoder import *
php	require_once('yocto_quadraturedecoder.php');
es	in HTML: <script src="../../lib/yocto_quadraturedecoder.js"></script> in node.js: require('yoctolib-es2017/yocto_quadraturedecoder.js');

Global functions

yFindQuadratureDecoder(**func**)

Retrieves a quadrature decoder for a given identifier.

yFindQuadratureDecoderInContext(**yctx**, **func**)

Retrieves a quadrature decoder for a given identifier in a YAPI context.

yFirstQuadratureDecoder()

Starts the enumeration of quadrature decoders currently accessible.

yFirstQuadratureDecoderInContext(**yctx**)

Starts the enumeration of quadrature decoders currently accessible.

YQuadratureDecoder methods

quadraturedecoder→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

quadraturedecoder→clearCache()

Invalidates the cache.

quadraturedecoder→describe()

Returns a short text that describes unambiguously the instance of the quadrature decoder in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

quadraturedecoder→get_advertisedValue()

Returns the current value of the quadrature decoder (no more than 6 characters).

quadraturedecoder→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in pas, as a floating point number.

quadraturedecoder→get_currentValue()

Returns the current value of the position, in pas, as a floating point number.

quadraturedecoder→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

quadraturedecoder→get_decoding()

Returns the current activation state of the quadrature decoder.

quadraturedecoder→get_errorMessage()

Returns the error message of the latest error with the quadrature decoder.

quadraturedecoder→get_errorType()

Returns the numerical error code of the latest error with the quadrature decoder.

quadraturedecoder→get_friendlyName()

Returns a global identifier of the quadrature decoder in the format `MODULE_NAME.FUNCTION_NAME`.

quadraturedecoder→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

quadraturedecoder→get_functionId()

Returns the hardware identifier of the quadrature decoder, without reference to the module.

quadraturedecoder→get_hardwareId()

Returns the unique hardware identifier of the quadrature decoder in the form `SERIAL.FUNCTIONID`.

quadraturedecoder→get_highestValue()

Returns the maximal value observed for the position since the device was started.

quadraturedecoder→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

quadraturedecoder→get_logicalName()

Returns the logical name of the quadrature decoder.

quadraturedecoder→get_lowestValue()

Returns the minimal value observed for the position since the device was started.

quadraturedecoder→get_module()

Gets the `YModule` object for the device on which the function is located.

quadraturedecoder→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

quadraturedecoder→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

quadraturedecoder→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

quadraturedecoder→get_resolution()

Returns the resolution of the measured values.

quadraturedecoder→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

quadraturedecoder→get_speed()

Returns the increments frequency, in Hz.

quadraturedecoder→get_unit()

Returns the measuring unit for the position.

quadraturedecoder→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

quadraturedecoder→isOnline()

Checks if the quadrature decoder is currently reachable, without raising any error.

quadraturedecoder→isOnline_async(**callback**, **context**)

Checks if the quadrature decoder is currently reachable, without raising any error (asynchronous version).

quadraturedecoder→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

quadraturedecoder→load(msValidity)

Preloads the quadrature decoder cache with a specified validity duration.

quadraturedecoder→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

quadraturedecoder→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

quadraturedecoder→load_async(msValidity, callback, context)

Preloads the quadrature decoder cache with a specified validity duration (asynchronous version).

quadraturedecoder→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

quadraturedecoder→nextQuadratureDecoder()

Continues the enumeration of quadrature decoders started using `yFirstQuadratureDecoder()`.

quadraturedecoder→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

quadraturedecoder→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

quadraturedecoder→set_currentValue(newval)

Changes the current expected position of the quadrature decoder.

quadraturedecoder→set_decoding(newval)

Changes the activation state of the quadrature decoder.

quadraturedecoder→set_highestValue(newval)

Changes the recorded maximal value observed.

quadraturedecoder→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

quadraturedecoder→set_logicalName(newval)

Changes the logical name of the quadrature decoder.

quadraturedecoder→set_lowestValue(newval)

Changes the recorded minimal value observed.

quadraturedecoder→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

quadraturedecoder→set_resolution(newval)

Changes the resolution of the measured physical values.

quadraturedecoder→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

quadraturedecoder→startDataLogger()

Starts the data logger on the device.

quadraturedecoder→stopDataLogger()

Stops the datalogger on the device.

quadraturedecoder→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

quadraturedecoder→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YQuadratureDecoder.FindQuadratureDecoder() YQuadratureDecoder yFindQuadratureDecoder() yFindQuadratureDecoder()

Retrieves a quadrature decoder for a given identifier.

function **FindQuadratureDecoder**(**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 quadrature decoder is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQuadratureDecoder.isOnline()` to test if the quadrature decoder is indeed online at a given time. In case of ambiguity when looking for a quadrature decoder 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the quadrature decoder

Returns :

a `YQuadratureDecoder` object allowing you to drive the quadrature decoder.

YQuadratureDecoder.FindQuadratureDecoderInContext() YQuadratureDecoder yFindQuadratureDecoderInContext()

Retrieves a quadrature decoder for a given identifier in a YAPI context.

function **FindQuadratureDecoderInContext**(**yctx**, **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 quadrature decoder is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQuadratureDecoder.isOnline()` to test if the quadrature decoder is indeed online at a given time. In case of ambiguity when looking for a quadrature decoder 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 :

yctx a YAPI context

func a string that uniquely characterizes the quadrature decoder

Returns :

a `YQuadratureDecoder` object allowing you to drive the quadrature decoder.

YQuadratureDecoder.FirstQuadratureDecoder() YQuadratureDecoder yFirstQuadratureDecoder() yFirstQuadratureDecoder()

Starts the enumeration of quadrature decoders currently accessible.

function **FirstQuadratureDecoder**( )

Use the method `YQuadratureDecoder.nextQuadratureDecoder()` to iterate on next quadrature decoders.

Returns :

a pointer to a `YQuadratureDecoder` object, corresponding to the first quadrature decoder currently online, or a `null` pointer if there are none.

YQuadratureDecoder.FirstQuadratureDecoderInContext() YQuadratureDecoder yFirstQuadratureDecoderInContext()

Starts the enumeration of quadrature decoders currently accessible.

function **FirstQuadratureDecoderInContext**(**yctx**)

Use the method `YQuadratureDecoder.nextQuadratureDecoder()` to iterate on next quadrature decoders.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YQuadratureDecoder` object, corresponding to the first quadrature decoder currently online, or a `null` pointer if there are none.

quadraturedecoder→calibrateFromPoints() YQuadratureDecoder
quadraturedecoder.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.

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.

quadraturedecoder→clearCache() YQuadratureDecoder
quadraturedecoder.clearCache()

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the quadrature decoder attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

quadraturedecoder→describe() YQuadratureDecoder
quadraturedecoder.describe()

Returns a short text that describes unambiguously the instance of the quadrature decoder in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the quadrature decoder (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

quadraturedecoder→get_advertisedValue() YQuadratureDecoder
quadraturedecoder→advertisedValue()
quadraturedecoder.get_advertisedValue()

Returns the current value of the quadrature decoder (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the quadrature decoder (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

quadraturedecoder→get_currentRawValue() YQuadratureDecoder
quadraturedecoder→currentRawValue()
quadraturedecoder.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in pas, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in pas, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

quadraturedecoder→get_currentValue() YQuadratureDecoder
quadraturedecoder→currentValue()
quadraturedecoder.get_currentValue()

Returns the current value of the position, in pas, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the position, in pas, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

quadraturedecoder→get_dataLogger() YQuadratureDecoder
quadraturedecoder→dataLogger()
quadraturedecoder.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

quadraturedecoder→get_decoding() YQuadratureDecoder
quadraturedecoder→decoding()
quadraturedecoder.get_decoding()

Returns the current activation state of the quadrature decoder.

function **get_decoding**( )

Returns :

either `Y_DECODING_OFF` or `Y_DECODING_ON`, according to the current activation state of the quadrature decoder

On failure, throws an exception or returns `Y_DECODING_INVALID`.

quadraturedecoder→get_errorMessage() YQuadratureDecoder
quadraturedecoder→errorMessage()
quadraturedecoder.get_errorMessage()

Returns the error message of the latest error with the quadrature decoder.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the quadrature decoder object

quadraturedecoder→get_errorType() YQuadratureDecoder
quadraturedecoder→errorType()
quadraturedecoder.get_errorType()

Returns the numerical error code of the latest error with the quadrature decoder.

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 the quadrature decoder object

quadraturedecoder→get_friendlyName() YQuadratureDecoder
quadraturedecoder→friendlyName()
quadraturedecoder.get_friendlyName()

Returns a global identifier of the quadrature decoder in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the quadrature decoder if they are defined, otherwise the serial number of the module and the hardware identifier of the quadrature decoder (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the quadrature decoder using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

quadraturedecoder→get_functionDescriptor() YQuadratureDecoder
quadraturedecoder→functionDescriptor()
quadraturedecoder.get_functionDescriptor()

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.

quadraturedecoder→get_functionId() YQuadratureDecoder
quadraturedecoder→functionId()
quadraturedecoder.get_functionId()

Returns the hardware identifier of the quadrature decoder, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the quadrature decoder (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

quadraturedecoder→get_hardwareId() YQuadratureDecoder
quadraturedecoder→hardwareId()
quadraturedecoder.get_hardwareId()

Returns the unique hardware identifier of the quadrature decoder in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quadrature decoder (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the quadrature decoder (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

quadraturedecoder→get_highestValue() YQuadratureDecoder
quadraturedecoder→highestValue()
quadraturedecoder.get_highestValue()

Returns the maximal value observed for the position since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the position since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

quadraturedecoder→get_logFrequency() YQuadratureDecoder
quadraturedecoder→logFrequency()
quadraturedecoder.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

quadraturedecoder→get_logicalName() YQuadratureDecoder
quadraturedecoder→logicalName()
quadraturedecoder.get_logicalName()

Returns the logical name of the quadrature decoder.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the quadrature decoder.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

quadraturedecoder→get_lowestValue() YQuadratureDecoder
quadraturedecoder→lowestValue()
quadraturedecoder.get_lowestValue()

Returns the minimal value observed for the position since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the position since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

quadraturedecoder→get_module() YQuadratureDecoder
quadraturedecoder→module()
quadraturedecoder.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

quadraturedecoder→get_recordedData() YQuadratureDecoder
quadraturedecoder→recordedData()
quadraturedecoder.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

quadraturedecoder→get_reportFrequency() YQuadratureDecoder
quadraturedecoder→reportFrequency()
quadraturedecoder.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

quadraturedecoder→get_resolution() YQuadratureDecoder
quadraturedecoder→resolution()
quadraturedecoder.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

quadraturedecoder→get_sensorState() YQuadratureDecoder
quadraturedecoder→sensorState()
quadraturedecoder.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :
an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

quadraturedecoder→get_speed()
quadraturedecoder→speed()
quadraturedecoder.get_speed()

Returns the increments frequency, in Hz.

function **get_speed**( )

Returns :
a floating point number corresponding to the increments frequency, in Hz

On failure, throws an exception or returns `Y_SPEED_INVALID`.

quadraturedecoder→get_unit() YQuadratureDecoder
quadraturedecoder→unit()
quadraturedecoder.get_unit()

Returns the measuring unit for the position.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the position

On failure, throws an exception or returns `Y_UNIT_INVALID`.

quadraturedecoder→get_userData() YQuadratureDecoder
quadraturedecoder→userData()
quadraturedecoder.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.

quadraturedecoder→isOnline() YQuadratureDecoder
quadraturedecoder.isOnline()

Checks if the quadrature decoder is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the quadrature decoder 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 quadrature decoder.

Returns :

`true` if the quadrature decoder can be reached, and `false` otherwise

quadraturedecoder→load()quadraturedecoder.load() YQuadratureDecoder

Preloads the quadrature decoder 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.

quadraturedecoder→loadAttribute() YQuadratureDecoder
quadraturedecoder.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

quadraturedecoder→loadCalibrationPoints() YQuadratureDecoder
quadraturedecoder.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→muteValueCallbacks() YQuadratureDecoder
quadraturedecoder.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→nextQuadratureDecoder() YQuadratureDecoder
quadraturedecoder.nextQuadratureDecoder()

Continues the enumeration of quadrature decoders started using `yFirstQuadratureDecoder()`.

function **nextQuadratureDecoder**( )

Returns :

a pointer to a `YQuadratureDecoder` object, corresponding to a quadrature decoder currently online, or a `null` pointer if there are no more quadrature decoders to enumerate.

quadraturedecoder→registerTimedReportCallback() YQuadratureDecoder quadraturedecoder.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

quadraturedecoder→registerValueCallback() YQuadratureDecoder
quadraturedecoder.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.

quadraturedecoder→set_currentValue() YQuadratureDecoder
quadraturedecoder→setCurrentValue()
quadraturedecoder.set_currentValue()

Changes the current expected position of the quadrature decoder.

```
function set_currentValue( newval)
```

Invoking this function implicitely activates the quadrature decoder.

Parameters :

newval a floating point number corresponding to the current expected position of the quadrature decoder

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→set_decoding() YQuadratureDecoder
quadraturedecoder→setDecoding()
quadraturedecoder.set_decoding()

Changes the activation state of the quadrature decoder.

function **set_decoding**(**newval**)

Parameters :

newval either `Y_DECODING_OFF` or `Y_DECODING_ON`, according to the activation state of the quadrature decoder

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→set_highestValue() YQuadratureDecoder
quadraturedecoder→setHighestValue()
quadraturedecoder.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.

quadraturedecoder→set_logFrequency() YQuadratureDecoder
quadraturedecoder→setLogFrequency()
quadraturedecoder.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→set_logicalName() YQuadratureDecoder
quadraturedecoder→setLogicalName()
quadraturedecoder.set_logicalName()

Changes the logical name of the quadrature decoder.

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 quadrature decoder.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→set_lowestValue() YQuadratureDecoder
quadraturedecoder→setLowestValue()
quadraturedecoder.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.

quadraturedecoder→set_reportFrequency() YQuadratureDecoder
quadraturedecoder→setReportFrequency()
quadraturedecoder.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→set_resolution() YQuadratureDecoder
quadraturedecoder→setResolution()
quadraturedecoder.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→set_userData() YQuadratureDecoder
quadraturedecoder→setUserData()
quadraturedecoder.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

quadraturedecoder→startDataLogger() YQuadratureDecoder
quadraturedecoder.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

quadraturedecoder→stopDataLogger() YQuadratureDecoder
quadraturedecoder.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

quadraturedecoder→unmuteValueCallbacks() YQuadratureDecoder
quadraturedecoder.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

quadraturedecoder→wait_async() YQuadratureDecoder
quadraturedecoder.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.54. RangeFinder function interface

The Yoctopuce class YRangeFinder allows you to use and configure Yoctopuce range finder sensors. It inherits from the YSensor class the core functions to read measurements, register callback functions, access the autonomous datalogger. This class adds the ability to easily perform a one-point linear calibration to compensate the effect of a glass or filter placed in front of the sensor.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_rangefinder.js'></script>
cpp	#include "yocto_rangefinder.h"
m	#import "yocto_rangefinder.h"
pas	uses yocto_rangefinder;
vb	yocto_rangefinder.vb
cs	yocto_rangefinder.cs
java	import com.yoctopuce.YoctoAPI.YRangeFinder;
uwp	import com.yoctopuce.YoctoAPI.YRangeFinder;
py	from yocto_rangefinder import *
php	require_once('yocto_rangefinder.php');
es	in HTML: <script src="../../lib/yocto_rangefinder.js"></script> in node.js: require('yoctolib-es2017/yocto_rangefinder.js');

Global functions

yFindRangeFinder(**func**)

Retrieves a range finder for a given identifier.

yFindRangeFinderInContext(**yctx**, **func**)

Retrieves a range finder for a given identifier in a YAPI context.

yFirstRangeFinder()

Starts the enumeration of range finders currently accessible.

yFirstRangeFinderInContext(**yctx**)

Starts the enumeration of range finders currently accessible.

`YRangeFinder` methods

rangefinder→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

rangefinder→cancelCoverGlassCalibrations()

Cancels the effect of previous hardware calibration procedures to compensate for cover glass, and restores factory settings.

rangefinder→clearCache()

Invalidates the cache.

rangefinder→describe()

Returns a short text that describes unambiguously the instance of the range finder in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

rangefinder→get_advertisedValue()

Returns the current value of the range finder (no more than 6 characters).

rangefinder→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in mm, as a floating point number.

rangefinder→get_currentTemperature()

Returns the current sensor temperature, as a floating point number.

rangefinder→get_currentValue()

Returns the current value of the range measured, in mm, as a floating point number.

rangefinder→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

rangefinder→get_errorMessage()

Returns the error message of the latest error with the range finder.

rangefinder→get_errorType()

Returns the numerical error code of the latest error with the range finder.

rangefinder→get_friendlyName()

Returns a global identifier of the range finder in the format `MODULE_NAME.FUNCTION_NAME`.

rangefinder→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

rangefinder→get_functionId()

Returns the hardware identifier of the range finder, without reference to the module.

rangefinder→get_hardwareCalibrationTemperature()

Returns the temperature at the time when the latest calibration was performed.

rangefinder→get_hardwareId()

Returns the unique hardware identifier of the range finder in the form `SERIAL.FUNCTIONID`.

rangefinder→get_highestValue()

Returns the maximal value observed for the range measured since the device was started.

rangefinder→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

rangefinder→get_logicalName()

Returns the logical name of the range finder.

rangefinder→get_lowestValue()

Returns the minimal value observed for the range measured since the device was started.

rangefinder→get_module()

Gets the `YModule` object for the device on which the function is located.

rangefinder→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

rangefinder→get_rangeFinderMode()

Returns the range finder running mode.

rangefinder→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

rangefinder→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

rangefinder→get_resolution()

Returns the resolution of the measured values.

rangefinder→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

rangefinder→get_unit()

Returns the measuring unit for the range measured.

rangefinder→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

rangefinder→isOnline()

Checks if the range finder is currently reachable, without raising any error.

rangefinder→isOnline_async(callback, context)

Checks if the range finder is currently reachable, without raising any error (asynchronous version).

rangefinder→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

rangefinder→load(msValidity)

Preloads the range finder cache with a specified validity duration.

rangefinder→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

rangefinder→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

rangefinder→load_async(msValidity, callback, context)

Preloads the range finder cache with a specified validity duration (asynchronous version).

rangefinder→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

rangefinder→nextRangeFinder()

Continues the enumeration of range finders started using `yFirstRangeFinder()`.

rangefinder→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

rangefinder→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

rangefinder→set_highestValue(newval)

Changes the recorded maximal value observed.

rangefinder→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

rangefinder→set_logicalName(newval)

Changes the logical name of the range finder.

rangefinder→set_lowestValue(newval)

Changes the recorded minimal value observed.

rangefinder→set_rangeFinderMode(newval)

Changes the rangefinder running mode, allowing you to put priority on precision, speed or maximum range.

rangefinder→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

rangefinder→set_resolution(newval)

Changes the resolution of the measured physical values.

rangefinder→set_unit(newval)

Changes the measuring unit for the measured range.

rangefinder→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

rangefinder→startDataLogger()

Starts the data logger on the device.

rangefinder→stopDataLogger()

Stops the datalogger on the device.

rangefinder→triggerOffsetCalibration(targetDist)

Triggers the hardware offset calibration of the distance sensor.

rangefinder→triggerSpadCalibration()

Triggers the photon detector hardware calibration.

rangefinder→triggerTemperatureCalibration()

Triggers a sensor calibration according to the current ambient temperature.

rangefinder→triggerXTalkCalibration(**targetDist**)

Triggers the hardware cross-talk calibration of the distance sensor.

rangefinder→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

rangefinder→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRangeFinder.FindRangeFinder() YRangeFinder yFindRangeFinder()yFindRangeFinder()

Retrieves a range finder for a given identifier.

function **FindRangeFinder**(**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 range finder is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRangeFinder.isOnline()` to test if the range finder is indeed online at a given time. In case of ambiguity when looking for a range finder 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the range finder

Returns :

a `YRangeFinder` object allowing you to drive the range finder.

YRangeFinder.FindRangeFinderInContext() YRangeFinder yFindRangeFinderInContext()

Retrieves a range finder for a given identifier in a YAPI context.

function **FindRangeFinderInContext**(**yctx**, **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 range finder is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRangeFinder.isOnline()` to test if the range finder is indeed online at a given time. In case of ambiguity when looking for a range finder 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 :

yctx a YAPI context

func a string that uniquely characterizes the range finder

Returns :

a `YRangeFinder` object allowing you to drive the range finder.

YRangeFinder.FirstRangeFinder() YRangeFinder yFirstRangeFinder()yFirstRangeFinder()

Starts the enumeration of range finders currently accessible.

function **FirstRangeFinder**( )

Use the method `YRangeFinder.nextRangeFinder()` to iterate on next range finders.

Returns :

a pointer to a `YRangeFinder` object, corresponding to the first range finder currently online, or a `null` pointer if there are none.

YRangeFinder.FirstRangeFinderInContext() YRangeFinder yFirstRangeFinderInContext()

Starts the enumeration of range finders currently accessible.

function **FirstRangeFinderInContext**(**yctx**)

Use the method `YRangeFinder.nextRangeFinder()` to iterate on next range finders.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YRangeFinder` object, corresponding to the first range finder currently online, or a `null` pointer if there are none.

rangefinder→calibrateFromPoints() YRangeFinder
rangefinder.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.

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.

rangefinder→cancelCoverGlassCalibrations() YRangeFinder
rangefinder.cancelCoverGlassCalibrations()

Cancels the effect of previous hardware calibration procedures to compensate for cover glass, and restores factory settings.

function **cancelCoverGlassCalibrations**( )

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.

rangefinder→clearCache()rangefinder.clearCache() YRangeFinder

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the range finder attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

rangefinder→describe()rangefinder.describe() YRangeFinder

Returns a short text that describes unambiguously the instance of the range finder in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the range finder (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

rangefinder→get_advertisedValue() YRangeFinder
rangefinder→advertisedValue()
rangefinder.get_advertisedValue()

Returns the current value of the range finder (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the range finder (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

rangefinder→get_currentRawValue() YRangeFinder
rangefinder→currentRawValue()
rangefinder.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in mm, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in mm, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

rangefinder→get_currentTemperature() YRangeFinder
rangefinder→currentTemperature()
rangefinder.get_currentTemperature()

Returns the current sensor temperature, as a floating point number.

function **get_currentTemperature**( )

Returns :
a floating point number corresponding to the current sensor temperature, as a floating point number

On failure, throws an exception or returns `Y_CURRENTTEMPERATURE_INVALID`.

rangefinder→get_currentValue() YRangeFinder
rangefinder→currentValue()
rangefinder.get_currentValue()

Returns the current value of the range measured, in mm, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the range measured, in mm, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

rangefinder→get_dataLogger() YRangeFinder
rangefinder→dataLogger()
rangefinder.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

rangefinder→get_errorMessage() YRangeFinder
rangefinder→errorMessage()
rangefinder.get_errorMessage()

Returns the error message of the latest error with the range finder.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the range finder object

rangefinder→get_errorType() YRangeFinder
rangefinder→errorType()rangefinder.get_errorType()

Returns the numerical error code of the latest error with the range finder.

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 the range finder object

rangefinder→get_friendlyName() YRangeFinder
rangefinder→friendlyName()
rangefinder.get_friendlyName()

Returns a global identifier of the range finder in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the range finder if they are defined, otherwise the serial number of the module and the hardware identifier of the range finder (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the range finder using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

rangefinder→get_functionDescriptor() YRangeFinder
rangefinder→functionDescriptor()
rangefinder.get_functionDescriptor()

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.

rangefinder→get_functionId() YRangeFinder
rangefinder→functionId()rangefinder.get_functionId()

Returns the hardware identifier of the range finder, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the range finder (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

rangefinder→get_hardwareCalibrationTemperature() YRangeFinder
rangefinder→hardwareCalibrationTemperature()
rangefinder.get_hardwareCalibrationTemperature()

Returns the temperature at the time when the latest calibration was performed.

function **get_hardwareCalibrationTemperature**( )

This function can be used to determine if a new calibration for ambient temperature is required.

Returns :

a temperature, as a floating point number. On failure, throws an exception or return YAPI_INVALID_DOUBLE.

rangefinder→get_hardwareId() YRangeFinder
rangefinder→hardwareId()
rangefinder.get_hardwareId()

Returns the unique hardware identifier of the range finder in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the range finder (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the range finder (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

rangefinder→get_highestValue() YRangeFinder
rangefinder→highestValue()
rangefinder.get_highestValue()

Returns the maximal value observed for the range measured since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the range measured since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

rangefinder→get_logFrequency() YRangeFinder
rangefinder→logFrequency()
rangefinder.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

rangefinder→get_logicalName() YRangeFinder
rangefinder→logicalName()
rangefinder.get_logicalName()

Returns the logical name of the range finder.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the range finder.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

rangefinder→get_lowestValue() YRangeFinder
rangefinder→lowestValue()
rangefinder.get_lowestValue()

Returns the minimal value observed for the range measured since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the range measured since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

rangefinder→get_module() YRangeFinder
rangefinder→module()rangefinder.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

rangefinder→get_rangeFinderMode() YRangeFinder
rangefinder→rangeFinderMode()
rangefinder.get_rangeFinderMode()

Returns the range finder running mode.

function **get_rangeFinderMode**( )

The rangefinder running mode allows you to put priority on precision, speed or maximum range.

Returns :

a value among `Y_RANGEFINDERMODE_DEFAULT`, `Y_RANGEFINDERMODE_LONG_RANGE`, `Y_RANGEFINDERMODE_HIGH_ACCURACY` and `Y_RANGEFINDERMODE_HIGH_SPEED` corresponding to the range finder running mode

On failure, throws an exception or returns `Y_RANGEFINDERMODE_INVALID`.

rangefinder→get_recordedData() YRangeFinder
rangefinder→recordedData()
rangefinder.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

rangefinder→get_reportFrequency() YRangeFinder
rangefinder→reportFrequency()
rangefinder.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

rangefinder→get_resolution() YRangeFinder
rangefinder→resolution()rangefinder.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

rangefinder→get_sensorState() YRangeFinder
rangefinder→sensorState()
rangefinder.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :
an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

rangefinder→get_unit() YRangeFinder
rangefinder→unit()rangefinder.get_unit()

Returns the measuring unit for the range measured.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the range measured

On failure, throws an exception or returns `Y_UNIT_INVALID`.

rangefinder→get_userData() YRangeFinder
rangefinder→userData()rangefinder.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.

rangefinder→isOnline()rangefinder.isOnline() YRangeFinder

Checks if the range finder is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the range finder 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 range finder.

Returns :

`true` if the range finder can be reached, and `false` otherwise

rangefinder→load()rangefinder.load() YRangeFinder

Preloads the range finder 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.

rangefinder→loadAttribute() YRangeFinder
rangefinder.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

rangefinder→loadCalibrationPoints() YRangeFinder
rangefinder.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→muteValueCallbacks() YRangeFinder
rangefinder.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→nextRangeFinder() YRangeFinder
rangefinder.nextRangeFinder()

Continues the enumeration of range finders started using `yFirstRangeFinder()`.

function **nextRangeFinder**( )

Returns :

a pointer to a `YRangeFinder` object, corresponding to a range finder currently online, or a `null` pointer if there are no more range finders to enumerate.

rangefinder→registerTimedReportCallback() YRangeFinder
rangefinder.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

rangefinder→registerValueCallback() YRangeFinder
rangefinder.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.

rangefinder→set_highestValue() YRangeFinder
rangefinder→setHighestValue()
rangefinder.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.

rangefinder→set_logFrequency() YRangeFinder
rangefinder→setLogFrequency()
rangefinder.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→set_logicalName() YRangeFinder
rangefinder→setLogicalName()
rangefinder.set_logicalName()

Changes the logical name of the range finder.

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 range finder.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→set_lowestValue() YRangeFinder
rangefinder→setLowestValue()
rangefinder.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.

rangefinder→set_rangeFinderMode() YRangeFinder
rangefinder→setRangeFinderMode()
rangefinder.set_rangeFinderMode()

Changes the rangefinder running mode, allowing you to put priority on precision, speed or maximum range.

function **set_rangeFinderMode**(**newval**)

Parameters :

newval a value among `Y_RANGEFINDERMODE_DEFAULT`, `Y_RANGEFINDERMODE_LONG_RANGE`, `Y_RANGEFINDERMODE_HIGH_ACCURACY` and `Y_RANGEFINDERMODE_HIGH_SPEED` corresponding to the rangefinder running mode, allowing you to put priority on precision, speed or maximum range

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→set_reportFrequency() YRangeFinder
rangefinder→setReportFrequency()
rangefinder.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→set_resolution() YRangeFinder
rangefinder→setResolution()
rangefinder.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→set_unit() YRangeFinder
rangefinder→setUnit()rangefinder.set_unit()

Changes the measuring unit for the measured range.

function **set_unit**(**newval**)

That unit is a string. String value can be `"` or `mm`. Any other value is ignored. Remember to call the `saveToFlash()` method of the module if the modification must be kept. WARNING: if a specific calibration is defined for the rangeFinder function, a unit system change will probably break it.

Parameters :

newval a string corresponding to the measuring unit for the measured range

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→set_userData() YRangeFinder
rangefinder→setUserData()
rangefinder.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

rangefinder→startDataLogger() YRangeFinder
rangefinder.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

rangefinder→stopDataLogger() YRangeFinder
rangefinder.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

rangefinder→triggerOffsetCalibration() YRangeFinder
rangefinder.triggerOffsetCalibration()

Triggers the hardware offset calibration of the distance sensor.

function **triggerOffsetCalibration**(**targetDist**)

This function is part of the calibration procedure to compensate for the the effect of a cover glass. Make sure to read the chapter about hardware calibration for details on the calibration procedure for proper results.

Parameters :

targetDist true distance of the calibration target, in mm or inches, depending on the unit selected in the device

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

rangefinder→triggerSpadCalibration() YRangeFinder
rangefinder.triggerSpadCalibration()

Triggers the photon detector hardware calibration.

function **triggerSpadCalibration**( )

This function is part of the calibration procedure to compensate for the the effect of a cover glass. Make sure to read the chapter about hardware calibration for details on the calibration procedure for proper results.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

rangefinder→triggerTemperatureCalibration() YRangeFinder
rangefinder.triggerTemperatureCalibration()

Triggers a sensor calibration according to the current ambient temperature.

function **triggerTemperatureCalibration**( )

That calibration process needs no physical interaction with the sensor. It is performed automatically at device startup, but it is recommended to start it again when the temperature delta since the latest calibration exceeds 8°C.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

rangefinder→triggerXTalkCalibration() YRangeFinder
rangefinder.triggerXTalkCalibration()

Triggers the hardware cross-talk calibration of the distance sensor.

function **triggerXTalkCalibration**(**targetDist**)

This function is part of the calibration procedure to compensate for the the effect of a cover glass. Make sure to read the chapter about hardware calibration for details on the calibration procedure for proper results.

Parameters :

targetDist true distance of the calibration target, in mm or inches, depending on the unit selected in the device

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

rangefinder→unmuteValueCallbacks() YRangeFinder
rangefinder.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

rangefinder→wait_async()rangefinder.wait_async() YRangeFinder

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.55. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_realtimeclock.js'></script>
cpp	#include "yocto_realtimeclock.h"
m	#import "yocto_realtimeclock.h"
pas	uses yocto_realtimeclock;
vb	yocto_realtimeclock.vb
cs	yocto_realtimeclock.cs
java	import com.yoctopuce.YoctoAPI.YRealTimeClock;
uwp	import com.yoctopuce.YoctoAPI.YRealTimeClock;
py	from yocto_realtimeclock import *
php	require_once('yocto_realtimeclock.php');
es	in HTML: <script src="../../lib/yocto_realtimeclock.js"></script> in node.js: require('yoctolib-es2017/yocto_realtimeclock.js');

Global functions

yFindRealTimeClock(**func**)

Retrieves a clock for a given identifier.

yFindRealTimeClockInContext(**yctx**, **func**)

Retrieves a clock for a given identifier in a YAPI context.

yFirstRealTimeClock()

Starts the enumeration of clocks currently accessible.

yFirstRealTimeClockInContext(**yctx**)

Starts the enumeration of clocks currently accessible.

`YRealTimeClock` methods

realtimeclock→clearCache()

Invalidates the cache.

realtimeclock→describe()

Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

realtimeclock→get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

realtimeclock→get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss".

realtimeclock→get_errorMessage()

Returns the error message of the latest error with the clock.

realtimeclock→get_errorType()

Returns the numerical error code of the latest error with the clock.

realtimeclock→get_friendlyName()

Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.

realtimeclock→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

realtimeclock→get_functionId()
Returns the hardware identifier of the clock, without reference to the module.

realtimeclock→get_hardwareId()
Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.

realtimeclock→get_logicalName()
Returns the logical name of the clock.

realtimeclock→get_module()
Gets the `YModule` object for the device on which the function is located.

realtimeclock→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

realtimeclock→get_timeSet()
Returns true if the clock has been set, and false otherwise.

realtimeclock→get_unixTime()
Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

realtimeclock→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

realtimeclock→get_utcOffset()
Returns the number of seconds between current time and UTC time (time zone).

realtimeclock→isOnline()
Checks if the clock is currently reachable, without raising any error.

realtimeclock→isOnline_async(**callback**, **context**)
Checks if the clock is currently reachable, without raising any error (asynchronous version).

realtimeclock→load(**msValidity**)
Preloads the clock cache with a specified validity duration.

realtimeclock→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

realtimeclock→load_async(**msValidity**, **callback**, **context**)
Preloads the clock cache with a specified validity duration (asynchronous version).

realtimeclock→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

realtimeclock→nextRealTimeClock()
Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

realtimeclock→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

realtimeclock→set_logicalName(**newval**)
Changes the logical name of the clock.

realtimeclock→set_unixTime(**newval**)
Changes the current time.

realtimeclock→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

realtimeclock→set_utcOffset(**newval**)
Changes the number of seconds between current time and UTC time (time zone).

realtimeclock→unmuteValueCallbacks()
Re-enables the propagation of every new advertised value to the parent hub.

realtimeclock→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRealTimeClock.FindRealTimeClock() YRealTimeClock yFindRealTimeClock()yFindRealTimeClock()

Retrieves a clock for a given identifier.

```
function FindRealTimeClock( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the clock is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRealTimeClock.isOnline()` to test if the clock is indeed online at a given time. In case of ambiguity when looking for a clock by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the clock

Returns :

a `YRealTimeClock` object allowing you to drive the clock.

YRealTimeClock.FindRealTimeClockInContext() YRealTimeClock yFindRealTimeClockInContext()

Retrieves a clock for a given identifier in a YAPI context.

function **FindRealTimeClockInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the clock is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRealTimeClock.isOnline()` to test if the clock is indeed online at a given time. In case of ambiguity when looking for a clock by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the clock

Returns :

a `YRealTimeClock` object allowing you to drive the clock.

YRealTimeClock.FirstRealTimeClock() YRealTimeClock yFirstRealTimeClock()yFirstRealTimeClock()

Starts the enumeration of clocks currently accessible.

```
function FirstRealTimeClock( )
```

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

Returns :

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a `null` pointer if there are none.

YRealTimeClock.FirstRealTimeClockInContext() yFirstRealTimeClockInContext()

YRealTimeClock

Starts the enumeration of clocks currently accessible.

function **FirstRealTimeClockInContext**(**yctx**)

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a `null` pointer if there are none.

realtimeclock→clearCache() YRealTimeClock
realtimeclock.clearCache()

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the clock attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

realtimeclock→describe()realtimeclock.describe() YRealTimeClock

Returns a short text that describes unambiguously the instance of the clock in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the clock (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

realtimeclock→get_advertisedValue() YRealTimeClock
realtimeclock→advertisedValue()
realtimeclock.get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the clock (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

realtimeclock→get_dateTime() YRealTimeClock
realtimeclock→dateTime()
realtimeclock.get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss".

function **get_dateTime**( )

Returns :
a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns `Y_DATETIME_INVALID`.

realtimeclock→get_errorMessage() YRealTimeClock
realtimeclock→errorMessage()
realtimeclock.get_errorMessage()

Returns the error message of the latest error with the clock.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the clock object

realtimeclock→get_errorType() YRealTimeClock
realtimeclock→errorType()
realtimeclock.get_errorType()

Returns the numerical error code of the latest error with the clock.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the clock object

realtimeclock→get_friendlyName() YRealTimeClock
realtimeclock→friendlyName()
realtimeclock.get_friendlyName()

Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the clock if they are defined, otherwise the serial number of the module and the hardware identifier of the clock (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the clock using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

realtimeclock→get_functionDescriptor() YRealTimeClock
realtimeclock→functionDescriptor()
realtimeclock.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

realtimeclock→get_functionId() YRealTimeClock
realtimeclock→functionId()
realtimeclock.get_functionId()

Returns the hardware identifier of the clock, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the clock (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

realtimeclock→get_hardwareId() YRealTimeClock
realtimeclock→hardwareId()
realtimeclock.get_hardwareId()

Returns the unique hardware identifier of the clock in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the clock (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the clock (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

realtimeclock→get_logicalName() YRealTimeClock
realtimeclock→logicalName()
realtimeclock.get_logicalName()

Returns the logical name of the clock.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the clock.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

realtimeclock→get_module() YRealTimeClock
realtimeclock→module()realtimeclock.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

realtimeclock→get_timeSet() YRealTimeClock
realtimeclock→timeSet()realtimeclock.get_timeSet()

Returns true if the clock has been set, and false otherwise.

function **get_timeSet**( )

Returns :

either `Y_TIMESET_FALSE` or `Y_TIMESET_TRUE`, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns `Y_TIMESET_INVALID`.

realtimeclock→get_unixTime() YRealTimeClock
realtimeclock→unixTime()
realtimeclock.get_unixTime()

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

function **get_unixTime**( )

Returns :
an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns `Y_UNIXTIME_INVALID`.

realtimeclock→get_userData() YRealTimeClock
realtimeclock→userData()
realtimeclock.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.

realtimeclock→get_utcOffset() YRealTimeClock
realtimeclock→utcOffset()
realtimeclock.get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

function **get_utcOffset**( )

Returns :
an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns `Y_UTCOFFSET_INVALID`.

realtimeclock→isOnline()realtimeclock.isOnline() YRealTimeClock

Checks if the clock is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

Returns :

`true` if the clock can be reached, and `false` otherwise

realtimeclock→load()realtimeclock.load() YRealTimeClock

Preloads the clock cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→loadAttribute() YRealTimeClock
realtimeclock.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

realtimeclock→muteValueCallbacks() YRealTimeClock
realtimeclock.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→nextRealTimeClock() YRealTimeClock
realtimeclock.nextRealTimeClock()

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

function **nextRealTimeClock**( )

Returns :

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a `null` pointer if there are no more clocks to enumerate.

realtimeclock→registerValueCallback() YRealTimeClock
realtimeclock.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.

realtimeclock→set_logicalName() YRealTimeClock
realtimeclock→setLogicalName()
realtimeclock.set_logicalName()

Changes the logical name of the clock.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the clock.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→set_unixTime() YRealTimeClock
realtimeclock→setUnixTime()
realtimeclock.set_unixTime()

Changes the current time.

function **set_unixTime**(**newval**)

Time is specifid in Unix format (number of elapsed seconds since Jan 1st, 1970).

Parameters :

newval an integer corresponding to the current time

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→set_userData() YRealTimeClock
realtimeclock→setUserData()
realtimeclock.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

realtimeclock→set_utcOffset() YRealTimeClock
realtimeclock→setUtcOffset()
realtimeclock.set_utcOffset()

Changes the number of seconds between current time and UTC time (time zone).

function **set_utcOffset**(**newval**)

The timezone is automatically rounded to the nearest multiple of 15 minutes.

Parameters :

newval an integer corresponding to the number of seconds between current time and UTC time (time zone)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→unmuteValueCallbacks() YRealTimeClock
realtimeclock.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→wait_async() YRealTimeClock
realtimeclock.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.56. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_refframe.js'></script>
cpp	#include "yocto_refframe.h"
m	#import "yocto_refframe.h"
pas	uses yocto_refframe;
vb	yocto_refframe.vb
cs	yocto_refframe.cs
java	import com.yoctopuce.YoctoAPI.YRefFrame;
uwp	import com.yoctopuce.YoctoAPI.YRefFrame;
py	from yocto_refframe import *
php	require_once('yocto_refframe.php');
es	in HTML: <script src="../../lib/yocto_refframe.js"></script> in node.js: require('yoctolib-es2017/yocto_refframe.js');

Global functions

yFindRefFrame(**func**)

Retrieves a reference frame for a given identifier.

yFindRefFrameInContext(**yctx**, **func**)

Retrieves a reference frame for a given identifier in a YAPI context.

yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

yFirstRefFrameInContext(**yctx**)

Starts the enumeration of reference frames currently accessible.

YRefFrame methods

refframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

refframe→clearCache()

Invalidates the cache.

refframe→describe()

Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

refframe→get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

refframe→get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

refframe→get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

refframe→get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

reframe→get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

reframe→get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

reframe→get_bearing()

Returns the reference bearing used by the compass.

reframe→get_calibrationState()

Returns the 3D sensor calibration state (Yocto-3D-V2 only).

reframe→get_errorMessage()

Returns the error message of the latest error with the reference frame.

reframe→get_errorType()

Returns the numerical error code of the latest error with the reference frame.

reframe→get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

reframe→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

reframe→get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

reframe→get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

reframe→get_logicalName()

Returns the logical name of the reference frame.

reframe→get_measureQuality()

Returns estimated quality of the orientation (Yocto-3D-V2 only).

reframe→get_module()

Gets the `YModule` object for the device on which the function is located.

reframe→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

reframe→get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

reframe→isOnline()

Checks if the reference frame is currently reachable, without raising any error.

reframe→isOnline_async(**callback**, **context**)

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reframe→load(**msValidity**)

Preloads the reference frame cache with a specified validity duration.

reframe→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

refframe→load_async(msValidity, callback, context)

Preloads the reference frame cache with a specified validity duration (asynchronous version).

refframe→more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

refframe→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

refframe→nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

refframe→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

refframe→save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

refframe→set_bearing(newval)

Changes the reference bearing used by the compass.

refframe→set_logicalName(newval)

Changes the logical name of the reference frame.

refframe→set_mountPosition(position, orientation)

Changes the compass and tilt sensor frame of reference.

refframe→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

refframe→start3DCalibration()

Initiates the sensors tridimensional calibration process.

refframe→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

refframe→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRefFrame.FindRefFrame() YRefFrame
yFindRefFrame()yFindRefFrame()

Retrieves a reference frame for a given identifier.

function **FindRefFrame**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the reference frame

Returns :

a `YRefFrame` object allowing you to drive the reference frame.

YRefFrame.FindRefFrameInContext() YRefFrame yFindRefFrameInContext()

Retrieves a reference frame for a given identifier in a YAPI context.

function **FindRefFrameInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the reference frame

Returns :

a `YRefFrame` object allowing you to drive the reference frame.

YRefFrame.FirstRefFrame() YRefFrame
yFirstRefFrame()yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

function **FirstRefFrame**( )

Use the method `YRefFrame.nextRefFrame()` to iterate on next reference frames.

Returns :

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

YRefFrame.FirstRefFrameInContext() YRefFrame
yFirstRefFrameInContext()

Starts the enumeration of reference frames currently accessible.

function **FirstRefFrameInContext**(**yctx**)

Use the method `YRefFrame.nextRefFrame()` to iterate on next reference frames.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

refframe→cancel3DCalibration() YRefFrame
refframe.cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

function **cancel3DCalibration**( )

On failure, throws an exception or returns a negative error code.

refframe→clearCache()refframe.clearCache() YRefFrame

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the reference frame attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

refframe→describe()refframe.describe() YRefFrame

Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the reference frame (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

refframe→get_3DCalibrationHint() YRefFrame
refframe→3DCalibrationHint()
refframe.get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

function **get_3DCalibrationHint**( )

Returns :
a character string.

refframe→get_3DCalibrationLogMsg() YRefFrame
refframe→3DCalibrationLogMsg()
refframe.get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

function **get_3DCalibrationLogMsg**( )

When no new message is available, returns an empty string.

Returns :
a character string.

refframe→get_3DCalibrationProgress() YRefFrame
refframe→3DCalibrationProgress()
refframe.get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

function **get_3DCalibrationProgress**( )

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_3DCalibrationStage() YRefFrame
refframe→3DCalibrationStage()
refframe.get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

function **get_3DCalibrationStage**( )

Returns :
an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress() YRefFrame
refframe→3DCalibrationStageProgress()
refframe.get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

function **get_3DCalibrationStageProgress**( )

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue() YRefFrame
refframe→advertisedValue()
refframe.get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the reference frame (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

refframe→get_bearing() YRefFrame
refframe→bearing()refframe.get_bearing()

Returns the reference bearing used by the compass.

function **get_bearing**( )

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

Returns :
a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns `Y_BEARING_INVALID`.

refframe→get_calibrationState() YRefFrame
refframe→calibrationState()
refframe.get_calibrationState()

Returns the 3D sensor calibration state (Yocto-3D-V2 only).

function **get_calibrationState**( )

This function returns an integer representing the calibration state of the 3 inertial sensors of the BNO055 chip, found in the Yocto-3D-V2. Hundredths show the calibration state of the accelerometer, tenths show the calibration state of the magnetometer while units show the calibration state of the gyroscope. For each sensor, the value 0 means no calibration and the value 3 means full calibration.

Returns :

an integer representing the calibration state of Yocto-3D-V2: 333 when fully calibrated, 0 when not calibrated at all.

On failure, throws an exception or returns a negative error code. For the Yocto-3D (V1), this function always return -3 (unsupported function).

refframe→get_errorMessage() YRefFrame
refframe→errorMessage()
refframe.get_errorMessage()

Returns the error message of the latest error with the reference frame.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the reference frame object

refframe→get_errorType() YRefFrame
refframe→errorType()refframe.get_errorType()

Returns the numerical error code of the latest error with the reference frame.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the reference frame object

refframe→get_friendlyName() YRefFrame
refframe→friendlyName()refframe.get_friendlyName()

Returns a global identifier of the reference frame in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the reference frame if they are defined, otherwise the serial number of the module and the hardware identifier of the reference frame (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the reference frame using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

refframe→get_functionDescriptor() YRefFrame
refframe→functionDescriptor()
refframe.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

refframe→get_functionId() YRefFrame
refframe→functionId()refframe.get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the reference frame (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

refframe→get_hardwareId() YRefFrame
refframe→hardwareId()refframe.get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the reference frame (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

refframe→get_logicalName() YRefFrame
refframe→logicalName()refframe.get_logicalName()

Returns the logical name of the reference frame.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the reference frame.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

refframe→get_measureQuality() YRefFrame
refframe→measureQuality()
refframe.get_measureQuality()

Returns estimated quality of the orientation (Yocto-3D-V2 only).

function **get_measureQuality**( )

This function returns an integer between 0 and 3 representing the degree of confidence of the position estimate. When the value is 3, the estimation is reliable. Below 3, one should expect sudden corrections, in particular for heading (`compass` function). The most frequent causes for values below 3 are magnetic interferences, and accelerations or rotations beyond the sensor range.

Returns :
an integer between 0 and 3 (3 when the measure is reliable)

On failure, throws an exception or returns a negative error code. For the Yocto-3D (V1), this function always return -3 (unsupported function).

refframe→get_module() YRefFrame
refframe→module()refframe.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

refframe→get_mountOrientation() YRefFrame
refframe→mountOrientation()
refframe.get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

function **get_mountOrientation**( )

Returns :

a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns Y_MOUNTORIENTATION_INVALID.

refframe→get_mountPosition() YRefFrame
refframe→mountPosition()
refframe.get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

function **get_mountPosition**( )

Returns :

a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns Y_MOUNTPOSITION_INVALID.

refframe→get_userData() YRefFrame
refframe→userData()reframe.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.

refframe→isOnline()refframe.isOnline() YRefFrame

Checks if the reference frame is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

Returns :

`true` if the reference frame can be reached, and `false` otherwise

refframe→load()refframe.load() YRefFrame

Preloads the reference frame cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→loadAttribute()refframe.loadAttribute() YRefFrame

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

refframe→more3DCalibration() YRefFrame
refframe.more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

function **more3DCalibration**( )

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process.

On failure, throws an exception or returns a negative error code.

refframe→muteValueCallbacks() YRefFrame
refframe.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→nextRefFrame()reframe.nextRefFrame() YRefFrame

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

function **nextRefFrame**( )

Returns :

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a `null` pointer if there are no more reference frames to enumerate.

refframe→registerValueCallback() YRefFrame
refframe.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.

refframe→save3DCalibration() YRefFrame
refframe.save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

function **save3DCalibration**( )

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted.

On failure, throws an exception or returns a negative error code.

refframe→set_bearing() YRefFrame
refframe→setBearing()refframe.set_bearing()

Changes the reference bearing used by the compass.

```
function set_bearing( newval)
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North.

Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction.

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→set_logicalName() YRefFrame
refframe→setLogicalName()
refframe.set_logicalName()

Changes the logical name of the reference frame.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the reference frame.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→set_mountPosition() YRefFrame
refframe→setMountPosition()
refframe.set_mountPosition()

Changes the compass and tilt sensor frame of reference.

function **set_mountPosition**(**position**, **orientation**)

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

Parameters :

position a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

orientation a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

refframe→set_userData() YRefFrame
refframe→setUserData()reframe.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

refframe→start3DCalibration() YRefFrame
refframe.start3DCalibration()

Initiates the sensors tridimensional calibration process.

function **start3DCalibration**( )

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors.

After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`.

On failure, throws an exception or returns a negative error code.

refframe→unmuteValueCallbacks() YRefFrame
refframe.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

reframe→wait_async()reframe.wait_async() YRefFrame

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.57. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

```
js    <script type='text/javascript' src='yocto_relay.js'></script>
cpp   #include "yocto_relay.h"
m     #import "yocto_relay.h"
pas   uses yocto_relay;
vb    yocto_relay.vb
cs    yocto_relay.cs
java  import com.yoctopuce.YoctoAPI.YRelay;
uwp   import com.yoctopuce.YoctoAPI.YRelay;
py    from yocto_relay import *
php   require_once('yocto_relay.php');
es    in HTML: <script src="../../lib/yocto_relay.js"></script>
      in node.js: require('yoctolib-es2017/yocto_relay.js');
```

Global functions
yFindRelay(**func**) Retrieves a relay for a given identifier.
yFindRelayInContext(**yctx**, **func**) Retrieves a relay for a given identifier in a YAPI context.
yFirstRelay() Starts the enumeration of relays currently accessible.
yFirstRelayInContext(**yctx**) Starts the enumeration of relays currently accessible.
YRelay methods
relay→clearCache() Invalidates the cache.
relay→delayedPulse(**ms_delay**, **ms_duration**) Schedules a pulse.
relay→describe() Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
relay→get_advertisedValue() Returns the current value of the relay (no more than 6 characters).
relay→get_countdown() Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.
relay→get_errorMessage() Returns the error message of the latest error with the relay.
relay→get_errorType() Returns the numerical error code of the latest error with the relay.

relay→get_friendlyName()

Returns a global identifier of the relay in the format MODULE_NAME.FUNCTION_NAME.

relay→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

relay→get_functionId()

Returns the hardware identifier of the relay, without reference to the module.

relay→get_hardwareId()

Returns the unique hardware identifier of the relay in the form SERIAL.FUNCTIONID.

relay→get_logicalName()

Returns the logical name of the relay.

relay→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

relay→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

relay→get_module()

Gets the YModule object for the device on which the function is located.

relay→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

relay→get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

relay→get_pulseTimer()

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

relay→get_state()

Returns the state of the relays (A for the idle position, B for the active position).

relay→get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

relay→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

relay→isOnline()

Checks if the relay is currently reachable, without raising any error.

relay→isOnline_async(callback, context)

Checks if the relay is currently reachable, without raising any error (asynchronous version).

relay→load(msValidity)

Preloads the relay cache with a specified validity duration.

relay→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

relay→load_async(msValidity, callback, context)

Preloads the relay cache with a specified validity duration (asynchronous version).

relay→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

relay→nextRelay()

Continues the enumeration of relays started using `yFirstRelay()`.

relay→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

relay→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

relay→set_logicalName(newval)

Changes the logical name of the relay.

relay→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

relay→set_maxTimeOnStateB(newval)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

relay→set_output(newval)

Changes the output state of the relays, when used as a simple switch (single throw).

relay→set_state(newval)

Changes the state of the relays (A for the idle position, B for the active position).

relay→set_stateAtPowerOn(newval)

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

relay→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

relay→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

relay→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRelay.FindRelay() YRelay yFindRelay()yFindRelay()

Retrieves a relay for a given identifier.

```
function FindRelay( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the relay is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRelay.isOnline()` to test if the relay is indeed online at a given time. In case of ambiguity when looking for a relay by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the relay

Returns :

a `YRelay` object allowing you to drive the relay.

YRelay.FindRelayInContext() YRelay yFindRelayInContext()

Retrieves a relay for a given identifier in a YAPI context.

function **FindRelayInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the relay

Returns :

a `YRelay` object allowing you to drive the relay.

YRelay.FirstRelay() YRelay yFirstRelay()yFirstRelay()

Starts the enumeration of relays currently accessible.

function **FirstRelay**( )

Use the method `YRelay.nextRelay()` to iterate on next relays.

Returns :

a pointer to a `YRelay` object, corresponding to the first relay currently online, or a `null` pointer if there are none.

YRelay.FirstRelayInContext() YRelay yFirstRelayInContext()

Starts the enumeration of relays currently accessible.

function **FirstRelayInContext**(**yctx**)

Use the method `YRelay.nextRelay()` to iterate on next relays.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YRelay` object, corresponding to the first relay currently online, or a `null` pointer if there are none.

relay→clearCache()relay.clearCache() YRelay

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the relay attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

relay→delayedPulse()relay.delayedPulse() YRelay

Schedules a pulse.

function **delayedPulse**(**ms_delay**, **ms_duration**)

Parameters :

ms_delay waiting time before the pulse, in millisecondes

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→describe()relay.describe() YRelay

Returns a short text that describes unambiguously the instance of the relay in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the relay (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

relay→get_advertisedValue() YRelay
relay→advertisedValue()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_countdown() YRelay
relay→countdown()relay.get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

function **get_countdown**( )

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

relay→get_errorMessage() YRelay
relay→errorMessage()relay.get_errorMessage()

Returns the error message of the latest error with the relay.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the relay object

relay→get_errorType() YRelay
relay→errorType()relay.get_errorType()

Returns the numerical error code of the latest error with the relay.

```
function get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the relay object

relay→get_friendlyName() YRelay
relay→friendlyName()relay.get_friendlyName()

Returns a global identifier of the relay in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the relay if they are defined, otherwise the serial number of the module and the hardware identifier of the relay (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the relay using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

relay→get_functionDescriptor() YRelay
relay→functionDescriptor()
relay.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

relay→get_functionId() YRelay
relay→functionId()relay.get_functionId()

Returns the hardware identifier of the relay, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the relay (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

relay→get_hardwareId() YRelay
relay→hardwareId()relay.get_hardwareId()

Returns the unique hardware identifier of the relay in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the relay (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the relay (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

relay→get_logicalName() YRelay
relay→logicalName()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_maxTimeOnStateA() YRelay
relay→maxTimeOnStateA()
relay.get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

function **get_maxTimeOnStateA**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

relay→get_maxTimeOnStateB() YRelay
relay→maxTimeOnStateB()
relay.get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

function **get_maxTimeOnStateB**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

relay→get_module() YRelay
relay→module()relay.get_module()

Gets the YModule object for the device on which the function is located.

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_output() YRelay
relay→output()relay.get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

function **get_output**( )

Returns :

either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.

relay→get_pulseTimer() YRelay
relay→pulseTimer()relay.get_pulseTimer()

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

function **get_pulseTimer**( )

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

relay→get_state() YRelay
relay→state()relay.get_state()

Returns the state of the relays (A for the idle position, B for the active position).

function **get_state**( )

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

relay→get_stateAtPowerOn() YRelay
relay→stateAtPowerOn()relay.get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

function **get_stateAtPowerOn**( )

Returns :

a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B` corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns `Y_STATEATPOWERON_INVALID`.

relay→get_userData() YRelay
relay→userData()relay.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.

relay→isOnline()relay.isOnline() YRelay

Checks if the relay is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the relay.

Returns :

`true` if the relay can be reached, and `false` otherwise

relay→load()relay.load() YRelay

Preloads the relay cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()relay.loadAttribute() YRelay

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

relay→muteValueCallbacks() YRelay
relay.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→nextRelay()relay.nextRelay() YRelay

Continues the enumeration of relays started using `yFirstRelay()`.

function **nextRelay**( )

Returns :

a pointer to a `YRelay` object, corresponding to a relay currently online, or a `null` pointer if there are no more relays to enumerate.

relay→pulse()relay.pulse() YRelay

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

function **pulse**(**ms_duration**)

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→registerValueCallback() YRelay
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() YRelay
relay→setLogicalName()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_maxTimeOnStateA() YRelay
relay→setMaxTimeOnStateA()
relay.set_maxTimeOnStateA()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

function **set_maxTimeOnStateA**(**newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateB() YRelay
relay→setMaxTimeOnStateB()
relay.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

function **set_maxTimeOnStateB**(**newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_output() YRelay
relay→setOutput()relay.set_output()

Changes the output state of the relays, when used as a simple switch (single throw).

function **set_output**(**newval**)

Parameters :

newval either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the relays, when used as a simple switch (single throw)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_state() YRelay
relay→setState()relay.set_state()

Changes the state of the relays (A for the idle position, B for the active position).

function **set_state**(**newval**)

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_stateAtPowerOn() YRelay
relay→setStateAtPowerOn()
relay.set_stateAtPowerOn()

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

function **set_stateAtPowerOn**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_userData() YRelay
relay→setUserData()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

relay→unmuteValueCallbacks() YRelay
relay.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→wait_async()relay.wait_async() YRelay

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.58. SegmentedDisplay function interface

The SegmentedDisplay class allows you to drive segmented displays.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_segmenteddisplay.js'></script>
cpp	#include "yocto_segmenteddisplay.h"
m	#import "yocto_segmenteddisplay.h"
pas	uses yocto_segmenteddisplay;
vb	yocto_segmenteddisplay.vb
cs	yocto_segmenteddisplay.cs
java	import com.yoctopuce.YoctoAPI.YSegmentedDisplay;
uwp	import com.yoctopuce.YoctoAPI.YSegmentedDisplay;
py	from yocto_segmenteddisplay import *
php	require_once('yocto_segmenteddisplay.php');
es	in HTML: <script src="../../lib/yocto_segmenteddisplay.js"></script> in node.js: require('yoctolib-es2017/yocto_segmenteddisplay.js');

Global functions

yFindSegmentedDisplay(**func**)

Retrieves a segmented display for a given identifier.

yFindSegmentedDisplayInContext(**yctx**, **func**)

Retrieves a segmented display for a given identifier in a YAPI context.

yFirstSegmentedDisplay()

Starts the enumeration of segmented displays currently accessible.

yFirstSegmentedDisplayInContext(**yctx**)

Starts the enumeration of segmented displays currently accessible.

`YSegmentedDisplay` methods

segmenteddisplay→clearCache()

Invalidates the cache.

segmenteddisplay→describe()

Returns a short text that describes unambiguously the instance of the segmented displays in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

segmenteddisplay→get_advertisedValue()

Returns the current value of the segmented displays (no more than 6 characters).

segmenteddisplay→get_displayedText()

Returns the text currently displayed on the screen.

segmenteddisplay→get_errorMessage()

Returns the error message of the latest error with the segmented displays.

segmenteddisplay→get_errorType()

Returns the numerical error code of the latest error with the segmented displays.

segmenteddisplay→get_friendlyName()

Returns a global identifier of the segmented displays in the format `MODULE_NAME.FUNCTION_NAME`.

segmenteddisplay→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

segmenteddisplay→get_functionId()

Returns the hardware identifier of the segmented displays, without reference to the module.

segmenteddisplay→get_hardwareId()

Returns the unique hardware identifier of the segmented displays in the form `SERIAL.FUNCTIONID`.

segmenteddisplay→get_logicalName()

Returns the logical name of the segmented displays.

segmenteddisplay→get_module()

Gets the `YModule` object for the device on which the function is located.

segmenteddisplay→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

segmenteddisplay→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

segmenteddisplay→isOnline()

Checks if the segmented displays is currently reachable, without raising any error.

segmenteddisplay→isOnline_async(callback, context)

Checks if the segmented displays is currently reachable, without raising any error (asynchronous version).

segmenteddisplay→load(msValidity)

Preloads the segmented displays cache with a specified validity duration.

segmenteddisplay→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

segmenteddisplay→load_async(msValidity, callback, context)

Preloads the segmented displays cache with a specified validity duration (asynchronous version).

segmenteddisplay→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

segmenteddisplay→nextSegmentedDisplay()

Continues the enumeration of segmented displays started using `yFirstSegmentedDisplay()`.

segmenteddisplay→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

segmenteddisplay→set_displayedText(newval)

Changes the text currently displayed on the screen.

segmenteddisplay→set_logicalName(newval)

Changes the logical name of the segmented displays.

segmenteddisplay→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

segmenteddisplay→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

segmenteddisplay→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YSegmentedDisplay.FindSegmentedDisplay() YSegmentedDisplay yFindSegmentedDisplay()yFindSegmentedDisplay()

Retrieves a segmented display for a given identifier.

```
function FindSegmentedDisplay( 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 segmented displays is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSegmentedDisplay.isOnline()` to test if the segmented displays is indeed online at a given time. In case of ambiguity when looking for a segmented display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the segmented displays

Returns :

a `YSegmentedDisplay` object allowing you to drive the segmented displays.

YSegmentedDisplay.FindSegmentedDisplayInContext() YSegmentedDisplay yFindSegmentedDisplayInContext()

Retrieves a segmented display for a given identifier in a YAPI context.

function **FindSegmentedDisplayInContext**(**yctx**, **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 segmented displays is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSegmentedDisplay.isOnline()` to test if the segmented displays is indeed online at a given time. In case of ambiguity when looking for a segmented display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the segmented displays

Returns :

a `YSegmentedDisplay` object allowing you to drive the segmented displays.

YSegmentedDisplay.FirstSegmentedDisplay() YSegmentedDisplay yFirstSegmentedDisplay()yFirstSegmentedDisplay()

Starts the enumeration of segmented displays currently accessible.

function **FirstSegmentedDisplay**( )

Use the method `YSegmentedDisplay.nextSegmentedDisplay()` to iterate on next segmented displays.

Returns :

a pointer to a `YSegmentedDisplay` object, corresponding to the first segmented displays currently online, or a `null` pointer if there are none.

YSegmentedDisplay.FirstSegmentedDisplayInContext() yFirstSegmentedDisplayInContext()

YSegmentedDisplay

Starts the enumeration of segmented displays currently accessible.

function **FirstSegmentedDisplayInContext**(**yctx**)

Use the method `YSegmentedDisplay.nextSegmentedDisplay()` to iterate on next segmented displays.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YSegmentedDisplay` object, corresponding to the first segmented displays currently online, or a `null` pointer if there are none.

segmenteddisplay→clearCache() YSegmentedDisplay
segmenteddisplay.clearCache()

Invalidates the cache.

```
function clearCache( )
```

Invalidates the cache of the segmented displays attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

segmenteddisplay→describe() YSegmentedDisplay
segmenteddisplay.describe()

Returns a short text that describes unambiguously the instance of the segmented displays in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the segmented displays (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

segmenteddisplay→get_advertisedValue() YSegmentedDisplay
segmenteddisplay→advertisedValue()
segmenteddisplay.get_advertisedValue()

Returns the current value of the segmented displays (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the segmented displays (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

segmenteddisplay→get_displayedText() YSegmentedDisplay
segmenteddisplay→displayedText()
segmenteddisplay.get_displayedText()

Returns the text currently displayed on the screen.

function **get_displayedText**( )

Returns :
a string corresponding to the text currently displayed on the screen

On failure, throws an exception or returns `Y_DISPLAYEDTEXT_INVALID`.

segmenteddisplay→get_errorMessage() YSegmentedDisplay
segmenteddisplay→errorMessage()
segmenteddisplay.get_errorMessage()

Returns the error message of the latest error with the segmented displays.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the segmented displays object

segmenteddisplay→get_errorType() YSegmentedDisplay
segmenteddisplay→errorType()
segmenteddisplay.get_errorType()

Returns the numerical error code of the latest error with the segmented displays.

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 the segmented displays object

segmenteddisplay→get_friendlyName() YSegmentedDisplay
segmenteddisplay→friendlyName()
segmenteddisplay.get_friendlyName()

Returns a global identifier of the segmented displays in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the segmented displays if they are defined, otherwise the serial number of the module and the hardware identifier of the segmented displays (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the segmented displays using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

segmenteddisplay→get_functionDescriptor() YSegmentedDisplay
segmenteddisplay→functionDescriptor()
segmenteddisplay.get_functionDescriptor()

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.

segmenteddisplay→get_functionId() YSegmentedDisplay
segmenteddisplay→functionId()
segmenteddisplay.get_functionId()

Returns the hardware identifier of the segmented displays, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the segmented displays (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

segmenteddisplay→get_hardwareId() YSegmentedDisplay
segmenteddisplay→hardwareId()
segmenteddisplay.get_hardwareId()

Returns the unique hardware identifier of the segmented displays in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the segmented displays (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the segmented displays (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

segmenteddisplay→get_logicalName() YSegmentedDisplay
segmenteddisplay→logicalName()
segmenteddisplay.get_logicalName()

Returns the logical name of the segmented displays.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the segmented displays.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

segmenteddisplay→get_module() YSegmentedDisplay
segmenteddisplay→module()
segmenteddisplay.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

segmenteddisplay→get_userData() YSegmentedDisplay
segmenteddisplay→userData()
segmenteddisplay.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.

segmenteddisplay→isOnline() YSegmentedDisplay
segmenteddisplay.isOnline()

Checks if the segmented displays is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the segmented displays 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 segmented displays.

Returns :

`true` if the segmented displays can be reached, and `false` otherwise

segmenteddisplay→load()segmenteddisplay.load() YSegmentedDisplay

Preloads the segmented displays 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.

segmenteddisplay→loadAttribute() YSegmentedDisplay
segmenteddisplay.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

segmenteddisplay→muteValueCallbacks() YSegmentedDisplay
segmenteddisplay.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

segmenteddisplay→nextSegmentedDisplay() YSegmentedDisplay
segmenteddisplay.nextSegmentedDisplay()

Continues the enumeration of segmented displays started using `yFirstSegmentedDisplay()`.

function **nextSegmentedDisplay**( )

Returns :

a pointer to a `YSegmentedDisplay` object, corresponding to a segmented display currently online, or a `null` pointer if there are no more segmented displays to enumerate.

segmenteddisplay→registerValueCallback() YSegmentedDisplay
segmenteddisplay.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.

segmenteddisplay→set_displayedText() YSegmentedDisplay
segmenteddisplay→setDisplayedText()
segmenteddisplay.set_displayedText()

Changes the text currently displayed on the screen.

function **set_displayedText**(**newval**)

Parameters :

newval a string corresponding to the text currently displayed on the screen

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

segmenteddisplay→set_logicalName() YSegmentedDisplay
segmenteddisplay→setLogicalName()
segmenteddisplay.set_logicalName()

Changes the logical name of the segmented displays.

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 segmented displays.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

segmenteddisplay→set_userData() YSegmentedDisplay
segmenteddisplay→setUserData()
segmenteddisplay.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

segmenteddisplay→unmuteValueCallbacks() YSegmentedDisplay
segmenteddisplay.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

segmenteddisplay→wait_async() YSegmentedDisplay
segmenteddisplay.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.59. Sensor function interface

The YSensor class is the parent class for all Yoctopuce sensors. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provide a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButton class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
uwp     import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
php     require_once('yocto_api.php');
es      in HTML: <script src="../../lib/yocto_api.js"></script>
        in node.js: require('yoctolib-es2017/yocto_api.js');
```

Global functions
yFindSensor(**func**) Retrieves a sensor for a given identifier.
yFindSensorInContext(**yctx**, **func**) Retrieves a sensor for a given identifier in a YAPI context.
yFirstSensor() Starts the enumeration of sensors currently accessible.
yFirstSensorInContext(**yctx**) Starts the enumeration of sensors currently accessible.
YSensor methods
sensor→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
sensor→clearCache() Invalidates the cache.
sensor→describe() Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
sensor→get_advertisedValue() Returns the current value of the sensor (no more than 6 characters).
sensor→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.
sensor→get_currentValue() Returns the current value of the measure, in the specified unit, as a floating point number.
sensor→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

sensor→get_errorMessage()

Returns the error message of the latest error with the sensor.

sensor→get_errorType()

Returns the numerical error code of the latest error with the sensor.

sensor→get_friendlyName()

Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.

sensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

sensor→get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

sensor→get_hardwareId()

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

sensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

sensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

sensor→get_logicalName()

Returns the logical name of the sensor.

sensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

sensor→get_module()

Gets the `YModule` object for the device on which the function is located.

sensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

sensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

sensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

sensor→get_resolution()

Returns the resolution of the measured values.

sensor→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

sensor→get_unit()

Returns the measuring unit for the measure.

sensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

sensor→isOnline()

Checks if the sensor is currently reachable, without raising any error.

sensor→isOnline_async(**callback**, **context**)

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

sensor→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

sensor→load(msValidity)

Preloads the sensor cache with a specified validity duration.

sensor→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

sensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

sensor→load_async(msValidity, callback, context)

Preloads the sensor cache with a specified validity duration (asynchronous version).

sensor→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

sensor→nextSensor()

Continues the enumeration of sensors started using `yFirstSensor()`.

sensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

sensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

sensor→set_highestValue(newval)

Changes the recorded maximal value observed.

sensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

sensor→set_logicalName(newval)

Changes the logical name of the sensor.

sensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

sensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

sensor→set_resolution(newval)

Changes the resolution of the measured physical values.

sensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

sensor→startDataLogger()

Starts the data logger on the device.

sensor→stopDataLogger()

Stops the datalogger on the device.

sensor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

sensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YSensor.FindSensor() YSensor
yFindSensor()yFindSensor()

Retrieves a sensor for a given identifier.

function **FindSensor**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the sensor

Returns :

a `YSensor` object allowing you to drive the sensor.

YSensor.FindSensorInContext() YSensor
yFindSensorInContext()

Retrieves a sensor for a given identifier in a YAPI context.

function **FindSensorInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the sensor

Returns :

a `YSensor` object allowing you to drive the sensor.

YSensor.FirstSensor() YSensor
yFirstSensor()yFirstSensor()

Starts the enumeration of sensors currently accessible.

function **FirstSensor**( )

Use the method `YSensor.nextSensor()` to iterate on next sensors.

Returns :

a pointer to a `YSensor` object, corresponding to the first sensor currently online, or a `null` pointer if there are none.

YSensor.FirstSensorInContext() YSensor yFirstSensorInContext()

Starts the enumeration of sensors currently accessible.

```
function FirstSensorInContext( yctx)
```

Use the method `YSensor.nextSensor()` to iterate on next sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YSensor` object, corresponding to the first sensor currently online, or a `null` pointer if there are none.

sensor→calibrateFromPoints() YSensor
sensor.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.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→clearCache()sensor.clearCache() YSensor

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

sensor→describe()sensor.describe() YSensor

Returns a short text that describes unambiguously the instance of the sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

sensor→get_advertisedValue() YSensor
sensor→advertisedValue()
sensor.get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

sensor→get_currentRawValue() YSensor
sensor→currentRawValue()
sensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

sensor→get_currentValue() YSensor
sensor→currentValue()sensor.get_currentValue()

Returns the current value of the measure, in the specified unit, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the measure, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

sensor→get_dataLogger() YSensor
sensor→dataLogger()sensor.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

sensor→get_errorMessage() YSensor
sensor→errorMessage()sensor.get_errorMessage()

Returns the error message of the latest error with the sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the sensor object

sensor→get_errorType() YSensor
sensor→errorType()sensor.get_errorType()

Returns the numerical error code of the latest error with the sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the sensor object

sensor→get_friendlyName() YSensor
sensor→friendlyName()sensor.get_friendlyName()

Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

sensor→get_functionDescriptor() YSensor
sensor→functionDescriptor()
sensor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

sensor→get_functionId() YSensor
sensor→functionId()sensor.get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

sensor→get_hardwareId() YSensor
sensor→hardwareId()sensor.get_hardwareId()

Returns the unique hardware identifier of the sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

sensor→get_highestValue() YSensor
sensor→highestValue()sensor.get_highestValue()

Returns the maximal value observed for the measure since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

sensor→get_logFrequency() YSensor
sensor→logFrequency()sensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

sensor→get_logicalName() YSensor
sensor→logicalName()sensor.get_logicalName()

Returns the logical name of the sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

sensor→get_lowestValue() YSensor
sensor→lowestValue()sensor.get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

sensor→get_module() YSensor
sensor→module()sensor.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

sensor→get_recordedData() YSensor
sensor→recordedData()sensor.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

sensor→get_reportFrequency() YSensor
sensor→reportFrequency()
sensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

sensor→get_resolution() YSensor
sensor→resolution()sensor.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

sensor→get_sensorState() YSensor
sensor→sensorState()sensor.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

sensor→get_unit() YSensor
sensor→unit()sensor.get_unit()

Returns the measuring unit for the measure.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

sensor→get_userData() YSensor
sensor→userData()sensor.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.

sensor→isOnline()sensor.isOnline() YSensor

Checks if the sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the sensor.

Returns :

`true` if the sensor can be reached, and `false` otherwise

sensor→isSensorReady()sensor.isSensorReady() YSensor

Checks if the sensor is currently able to provide an up-to-date measure.

function **isSensorReady**( )

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

Returns :

`true` if the sensor can provide an up-to-date measure, and `false` otherwise

sensor→load()sensor.load() YSensor

Preloads the sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→loadAttribute()sensor.loadAttribute() YSensor

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

sensor→loadCalibrationPoints() YSensor
sensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→muteValueCallbacks() YSensor
sensor.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→nextSensor()sensor.nextSensor() YSensor

Continues the enumeration of sensors started using `yFirstSensor()`.

function **nextSensor**( )

Returns :

a pointer to a `YSensor` object, corresponding to a sensor currently online, or a `null` pointer if there are no more sensors to enumerate.

sensor→registerTimedReportCallback() YSensor
sensor.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

sensor→registerValueCallback() YSensor
sensor.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.

sensor→set_highestValue() YSensor
sensor→setHighestValue()sensor.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.

sensor→set_logFrequency() YSensor
sensor→setLogFrequency()
sensor.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_logicalName() YSensor
sensor→setLogicalName()sensor.set_logicalName()

Changes the logical name of the sensor.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_lowestValue() YSensor
sensor→setLowestValue()sensor.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.

sensor→set_reportFrequency() YSensor
sensor→setReportFrequency()
sensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_resolution() YSensor
sensor→setResolution()sensor.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_userData() YSensor
sensor→setUserData()sensor.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

sensor→startDataLogger()sensor.startDataLogger() YSensor

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

sensor→stopDataLogger()sensor.stopDataLogger() YSensor

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

sensor→unmuteValueCallbacks() YSensor
sensor.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→wait_async()sensor.wait_async() YSensor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.60. SerialPort function interface

The SerialPort function interface allows you to fully drive a Yoctopuce serial port, to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_serialport.js'></script>
cpp     #include "yocto_serialport.h"
m       #import "yocto_serialport.h"
pas     uses yocto_serialport;
vb      yocto_serialport.vb
cs      yocto_serialport.cs
java    import com.yoctopuce.YoctoAPI.YSerialPort;
uwp     import com.yoctopuce.YoctoAPI.YSerialPort;
py      from yocto_serialport import *
php     require_once('yocto_serialport.php');
es      in HTML: <script src="../../lib/yocto_serialport.js"></script>
        in node.js: require('yoctolib-es2017/yocto_serialport.js');
```

Global functions
yFindSerialPort(**func**) Retrieves a serial port for a given identifier.
yFindSerialPortInContext(**yctx**, **func**) Retrieves a serial port for a given identifier in a YAPI context.
yFirstSerialPort() Starts the enumeration of serial ports currently accessible.
yFirstSerialPortInContext(**yctx**) Starts the enumeration of serial ports currently accessible.
`YSerialPort` methods
serialport→clearCache() Invalidates the cache.
serialport→describe() Returns a short text that describes unambiguously the instance of the serial port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
serialport→get_CTS() Reads the level of the CTS line.
serialport→get_advertisedValue() Returns the current value of the serial port (no more than 6 characters).
serialport→get_currentJob() Returns the name of the job file currently in use.
serialport→get_errCount() Returns the total number of communication errors detected since last reset.
serialport→get_errorMessage() Returns the error message of the latest error with the serial port.
serialport→get_errorType() Returns the numerical error code of the latest error with the serial port.
serialport→get_friendlyName()

Returns a global identifier of the serial port in the format MODULE_NAME.FUNCTION_NAME.

serialport→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

serialport→get_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

serialport→get_hardwareId()

Returns the unique hardware identifier of the serial port in the form SERIAL.FUNCTIONID.

serialport→get_lastMsg()

Returns the latest message fully received (for Line, Frame and Modbus protocols).

serialport→get_logicalName()

Returns the logical name of the serial port.

serialport→get_module()

Gets the YModule object for the device on which the function is located.

serialport→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

serialport→get_protocol()

Returns the type of protocol used over the serial line, as a string.

serialport→get_rxCount()

Returns the total number of bytes received since last reset.

serialport→get_rxMsgCount()

Returns the total number of messages received since last reset.

serialport→get_serialMode()

Returns the serial port communication parameters, as a string such as "9600,8N1".

serialport→get_startupJob()

Returns the job file to use when the device is powered on.

serialport→get_txCount()

Returns the total number of bytes transmitted since last reset.

serialport→get_txMsgCount()

Returns the total number of messages send since last reset.

serialport→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

serialport→get_voltageLevel()

Returns the voltage level used on the serial line.

serialport→isOnline()

Checks if the serial port is currently reachable, without raising any error.

serialport→isOnline_async(**callback**, **context**)

Checks if the serial port is currently reachable, without raising any error (asynchronous version).

serialport→load(**msValidity**)

Preloads the serial port cache with a specified validity duration.

serialport→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

serialport→load_async(**msValidity**, **callback**, **context**)

Preloads the serial port cache with a specified validity duration (asynchronous version).

serialport→modbusReadBits(**slaveNo**, **pduAddr**, **nBits**)

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

serialport→modbusReadInputBits(slaveNo, pduAddr, nBits)

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

serialport→modbusReadInputRegisters(slaveNo, pduAddr, nWords)

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

serialport→modbusReadRegisters(slaveNo, pduAddr, nWords)

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

serialport→modbusWriteAndReadRegisters(slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

serialport→modbusWriteBit(slaveNo, pduAddr, value)

Sets a single internal bit (or coil) on a MODBUS serial device.

serialport→modbusWriteBits(slaveNo, pduAddr, bits)

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

serialport→modbusWriteRegister(slaveNo, pduAddr, value)

Sets a single internal register (or holding register) on a MODBUS serial device.

serialport→modbusWriteRegisters(slaveNo, pduAddr, values)

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

serialport→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

serialport→nextSerialPort()

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

serialport→queryLine(query, maxWait)

Sends a text line query to the serial port, and reads the reply, if any.

serialport→queryMODBUS(slaveNo, pduBytes)

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

serialport→readArray(nChars)

Reads data from the receive buffer as a list of bytes, starting at current stream position.

serialport→readBin(nChars)

Reads data from the receive buffer as a binary buffer, starting at current stream position.

serialport→readByte()

Reads one byte from the receive buffer, starting at current stream position.

serialport→readHex(nBytes)

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

serialport→readLine()

Reads a single line (or message) from the receive buffer, starting at current stream position.

serialport→readMessages(pattern, maxWait)

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

serialport→readStr(nChars)

Reads data from the receive buffer as a string, starting at current stream position.

serialport→read_avail()

Returns the number of bytes available to read in the input buffer starting from the current absolute stream position pointer of the API object.

serialport→read_seek(absPos)

Changes the current internal stream position to the specified value.

serialport→read_tell()

Returns the current absolute stream position pointer of the API object.

serialport→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

serialport→reset()

Clears the serial port buffer and resets counters to zero.

serialport→selectJob(jobfile)

Load and start processing the specified job file.

serialport→set_RTS(val)

Manually sets the state of the RTS line.

serialport→set_currentJob(newval)

Changes the job to use when the device is powered on.

serialport→set_logicalName(newval)

Changes the logical name of the serial port.

serialport→set_protocol(newval)

Changes the type of protocol used over the serial line.

serialport→set_serialMode(newval)

Changes the serial port communication parameters, with a string such as "9600,8N1".

serialport→set_startupJob(newval)

Changes the job to use when the device is powered on.

serialport→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

serialport→set_voltageLevel(newval)

Changes the voltage type used on the serial line.

serialport→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

serialport→uploadJob(jobfile, jsonDef)

Saves the job definition string (JSON data) into a job file.

serialport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

serialport→writeArray(byteList)

Sends a byte sequence (provided as a list of bytes) to the serial port.

serialport→writeBin(buff)

Sends a binary buffer to the serial port, as is.

serialport→writeByte(code)

Sends a single byte to the serial port.

serialport→writeHex(hexString)

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

serialport→writeLine(text)

Sends an ASCII string to the serial port, followed by a line break (CR LF).

serialport→writeMODBUS(hexString)

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

serialport→writeStr(text)

Sends an ASCII string to the serial port, as is.

YSerialPort.FindSerialPort() YSerialPort
yFindSerialPort()yFindSerialPort()

Retrieves a serial port for a given identifier.

```
function FindSerialPort( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the serial port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSerialPort.isOnline()` to test if the serial port is indeed online at a given time. In case of ambiguity when looking for a serial port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the serial port

Returns :

a `YSerialPort` object allowing you to drive the serial port.

YSerialPort.FindSerialPortInContext() YSerialPort yFindSerialPortInContext()

Retrieves a serial port for a given identifier in a YAPI context.

function **FindSerialPortInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the serial port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSerialPort.isOnline()` to test if the serial port is indeed online at a given time. In case of ambiguity when looking for a serial port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the serial port

Returns :

a `YSerialPort` object allowing you to drive the serial port.

YSerialPort.FirstSerialPort() YSerialPort yFirstSerialPort()yFirstSerialPort()

Starts the enumeration of serial ports currently accessible.

function **FirstSerialPort**( )

Use the method `YSerialPort.nextSerialPort()` to iterate on next serial ports.

Returns :

a pointer to a `YSerialPort` object, corresponding to the first serial port currently online, or a `null` pointer if there are none.

YSerialPort.FirstSerialPortInContext() YSerialPort yFirstSerialPortInContext()

Starts the enumeration of serial ports currently accessible.

function **FirstSerialPortInContext**(**yctx**)

Use the method `YSerialPort.nextSerialPort()` to iterate on next serial ports.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YSerialPort` object, corresponding to the first serial port currently online, or a `null` pointer if there are none.

serialport→clearCache()serialport.clearCache() YSerialPort

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the serial port attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

serialport→describe()serialport.describe() YSerialPort

Returns a short text that describes unambiguously the instance of the serial port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the serial port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

serialport→get_CTS() YSerialPort
serialport→CTS()serialport.get_CTS()

Reads the level of the CTS line.

function **get_CTS**( )

The CTS line is usually driven by the RTS signal of the connected serial device.

Returns :
1 if the CTS line is high, 0 if the CTS line is low.

On failure, throws an exception or returns a negative error code.

serialport→get_advertisedValue() YSerialPort
serialport→advertisedValue()
serialport.get_advertisedValue()

Returns the current value of the serial port (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the serial port (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

serialport→get_currentJob() YSerialPort
serialport→currentJob()serialport.get_currentJob()

Returns the name of the job file currently in use.

function **get_currentJob**( )

Returns :
a string corresponding to the name of the job file currently in use

On failure, throws an exception or returns `Y_CURRENTJOB_INVALID`.

serialport→get_errCount() YSerialPort
serialport→errCount()serialport.get_errCount()

Returns the total number of communication errors detected since last reset.

function **get_errCount**( )

Returns :
an integer corresponding to the total number of communication errors detected since last reset

On failure, throws an exception or returns `Y_ERRCOUNT_INVALID`.

serialport→get_errorMessage() YSerialPort
serialport→errorMessage()
serialport.get_errorMessage()

Returns the error message of the latest error with the serial port.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the serial port object

serialport→get_errorType() YSerialPort
serialport→errorType()serialport.get_errorType()

Returns the numerical error code of the latest error with the serial port.

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 the serial port object

serialport→get_friendlyName() YSerialPort
serialport→friendlyName()
serialport.get_friendlyName()

Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the serial port if they are defined, otherwise the serial number of the module and the hardware identifier of the serial port (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the serial port using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

serialport→get_functionDescriptor() YSerialPort
serialport→functionDescriptor()
serialport.get_functionDescriptor()

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.

serialport→get_functionId() YSerialPort
serialport→functionId()serialport.get_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the serial port (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

serialport→get_hardwareId() YSerialPort
serialport→hardwareId()serialport.get_hardwareId()

Returns the unique hardware identifier of the serial port in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the serial port (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the serial port (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

serialport→get_lastMsg() YSerialPort
serialport→lastMsg()serialport.get_lastMsg()

Returns the latest message fully received (for Line, Frame and Modbus protocols).

function **get_lastMsg**( )

Returns :
a string corresponding to the latest message fully received (for Line, Frame and Modbus protocols)

On failure, throws an exception or returns `Y_LASTMSG_INVALID`.

serialport→get_logicalName() YSerialPort
serialport→logicalName()
serialport.get_logicalName()

Returns the logical name of the serial port.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the serial port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

serialport→get_module() YSerialPort
serialport→module()serialport.get_module()

Gets the YModule object for the device on which the function is located.

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

serialport→get_protocol() YSerialPort
serialport→protocol()serialport.get_protocol()

Returns the type of protocol used over the serial line, as a string.

```
function get_protocol( )
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Wiegand-ASCII" for Wiegand messages in ASCII mode, "Wiegand-26","Wiegand-34", etc for Wiegand messages in byte mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

Returns :
a string corresponding to the type of protocol used over the serial line, as a string

On failure, throws an exception or returns `Y_PROTOCOL_INVALID`.

serialport→get_rxCount() YSerialPort
serialport→rxCount()serialport.get_rxCount()

Returns the total number of bytes received since last reset.

function **get_rxCount**( )

Returns :
an integer corresponding to the total number of bytes received since last reset

On failure, throws an exception or returns `Y_RXCOUNT_INVALID`.

serialport→get_rxMsgCount() YSerialPort
serialport→rxMsgCount()
serialport.get_rxMsgCount()

Returns the total number of messages received since last reset.

function **get_rxMsgCount**( )

Returns :
an integer corresponding to the total number of messages received since last reset

On failure, throws an exception or returns `Y_RXMSGCOUNT_INVALID`.

serialport→get_serialMode() YSerialPort
serialport→serialMode()serialport.get_serialMode()

Returns the serial port communication parameters, as a string such as "9600,8N1".

function **get_serialMode**( )

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix is included if flow control is active: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Returns :
a string corresponding to the serial port communication parameters, as a string such as "9600,8N1"

On failure, throws an exception or returns `Y_SERIALMODE_INVALID`.

serialport→get_startupJob() YSerialPort
serialport→startupJob()serialport.get_startupJob()

Returns the job file to use when the device is powered on.

function **get_startupJob**( )

Returns :
a string corresponding to the job file to use when the device is powered on

On failure, throws an exception or returns `Y_STARTUPJOB_INVALID`.

serialport→get_txCount() YSerialPort
serialport→txCount()serialport.get_txCount()

Returns the total number of bytes transmitted since last reset.

function **get_txCount**( )

Returns :
an integer corresponding to the total number of bytes transmitted since last reset

On failure, throws an exception or returns `Y_TXCOUNT_INVALID`.

serialport→get_txMsgCount() YSerialPort
serialport→txMsgCount()serialport.get_txMsgCount()

Returns the total number of messages send since last reset.

function **get_txMsgCount**( )

Returns :
an integer corresponding to the total number of messages send since last reset

On failure, throws an exception or returns `Y_TXMSGCOUNT_INVALID`.

serialport→get_userData() YSerialPort
serialport→userData()serialport.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.

serialport→get_voltageLevel() YSerialPort
serialport→voltageLevel()
serialport.get_voltageLevel()

Returns the voltage level used on the serial line.

function **get_voltageLevel**( )

Returns :

a value among Y_VOLTAGELEVEL_OFF, Y_VOLTAGELEVEL_TTL3V, Y_VOLTAGELEVEL_TTL3VR, Y_VOLTAGELEVEL_TTL5V, Y_VOLTAGELEVEL_TTL5VR, Y_VOLTAGELEVEL_RS232 and Y_VOLTAGELEVEL_RS485 corresponding to the voltage level used on the serial line

On failure, throws an exception or returns Y_VOLTAGELEVEL_INVALID.

serialport→isOnline()serialport.isOnline() YSerialPort

Checks if the serial port is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the serial port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the serial port.

Returns :

`true` if the serial port can be reached, and `false` otherwise

serialport→load()serialport.load() YSerialPort

Preloads the serial port cache with a specified validity duration.

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.

serialport→loadAttribute()serialport.loadAttribute() YSerialPort

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

serialport→modbusReadBits() YSerialPort
serialport.modbusReadBits()

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

function **modbusReadBits**(**slaveNo**, **pduAddr**, **nBits**)

This method uses the MODBUS function code 0x01 (Read Coils).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first bit/coil to read (zero-based)

nBits the number of bits/coils to read

Returns :

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

serialport→modbusReadInputBits() YSerialPort
serialport.modbusReadInputBits()

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

function **modbusReadInputBits**(**slaveNo**, **pduAddr**, **nBits**)

This method uses the MODBUS function code 0x02 (Read Discrete Inputs).

Parameters :

slaveNo the address of the slave MODBUS device to query
pduAddr the relative address of the first bit/input to read (zero-based)
nBits the number of bits/inputs to read

Returns :

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

serialport→modbusReadInputRegisters() YSerialPort
serialport.modbusReadInputRegisters()

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

function **modbusReadInputRegisters**(**slaveNo**, **pduAddr**, **nWords**)

This method uses the MODBUS function code 0x04 (Read Input Registers).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first input register to read (zero-based)

nWords the number of input registers to read

Returns :

a vector of integers, each corresponding to one 16-bit input value.

On failure, throws an exception or returns an empty array.

serialport→modbusReadRegisters() YSerialPort
serialport.modbusReadRegisters()

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

function **modbusReadRegisters**(**slaveNo**, **pduAddr**, **nWords**)

This method uses the MODBUS function code 0x03 (Read Holding Registers).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first holding register to read (zero-based)

nWords the number of holding registers to read

Returns :

a vector of integers, each corresponding to one 16-bit register value.

On failure, throws an exception or returns an empty array.

serialport→modbusWriteAndReadRegisters() YSerialPort
serialport.modbusWriteAndReadRegisters()

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

function **modbusWriteAndReadRegisters**(**slaveNo**, **pduWriteAddr**, **values**, **pduReadAddr**, **nReadWords**)

This method uses the MODBUS function code 0x17 (Read/Write Multiple Registers).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduWriteAddr	the relative address of the first internal register to set (zero-based)
values	the vector of 16 bit values to set
pduReadAddr	the relative address of the first internal register to read (zero-based)
nReadWords	the number of 16 bit values to read

Returns :

a vector of integers, each corresponding to one 16-bit register value read.

On failure, throws an exception or returns an empty array.

serialport→modbusWriteBit() YSerialPort
serialport.modbusWriteBit()

Sets a single internal bit (or coil) on a MODBUS serial device.

function **modbusWriteBit**(**slaveNo**, **pduAddr**, **value**)

This method uses the MODBUS function code 0x05 (Write Single Coil).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the bit/coil to set (zero-based)

value the value to set (0 for OFF state, non-zero for ON state)

Returns :

the number of bits/coils affected on the device (1)

On failure, throws an exception or returns zero.

serialport→modbusWriteBits() YSerialPort
serialport.modbusWriteBits()

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

function **modbusWriteBits**(**slaveNo**, **pduAddr**, **bits**)

This method uses the MODBUS function code 0x0f (Write Multiple Coils).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the first bit/coil to set (zero-based)

bits the vector of bits to be set (one integer per bit)

Returns :

the number of bits/coils affected on the device

On failure, throws an exception or returns zero.

serialport→modbusWriteRegister() YSerialPort
serialport.modbusWriteRegister()

Sets a single internal register (or holding register) on a MODBUS serial device.

function **modbusWriteRegister**(**slaveNo**, **pduAddr**, **value**)

This method uses the MODBUS function code 0x06 (Write Single Register).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the register to set (zero-based)

value the 16 bit value to set

Returns :

the number of registers affected on the device (1)

On failure, throws an exception or returns zero.

serialport→modbusWriteRegisters() YSerialPort
serialport.modbusWriteRegisters()

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

function **modbusWriteRegisters**(**slaveNo**, **pduAddr**, **values**)

This method uses the MODBUS function code 0x10 (Write Multiple Registers).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the first internal register to set (zero-based)

values the vector of 16 bit values to set

Returns :

the number of registers affected on the device

On failure, throws an exception or returns zero.

serialport→muteValueCallbacks() YSerialPort
serialport.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→nextSerialPort()serialport.nextSerialPort() YSerialPort

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

function **nextSerialPort**( )

Returns :

a pointer to a `YSerialPort` object, corresponding to a serial port currently online, or a `null` pointer if there are no more serial ports to enumerate.

serialport→queryLine()serialport.queryLine() YSerialPort

Sends a text line query to the serial port, and reads the reply, if any.

function **queryLine**(**query**, **maxWait**)

This function is intended to be used when the serial port is configured for 'Line' protocol.

Parameters :

query the line query to send (without CR/LF)

maxWait the maximum number of milliseconds to wait for a reply.

Returns :

the next text line received after sending the text query, as a string. Additional lines can be obtained by calling readLine or readMessages.

On failure, throws an exception or returns an empty array.

serialport→queryMODBUS() YSerialPort
serialport.queryMODBUS()

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

function **queryMODBUS**(**slaveNo**, **pduBytes**)

The message is the PDU, provided as a vector of bytes.

Parameters :

slaveNo the address of the slave MODBUS device to query

pduBytes the message to send (PDU), as a vector of bytes. The first byte of the PDU is the MODBUS function code.

Returns :

the received reply, as a vector of bytes.

On failure, throws an exception or returns an empty array (or a MODBUS error reply).

serialport→readArray()serialport.readArray() YSerialPort

Reads data from the receive buffer as a list of bytes, starting at current stream position.

```
function readArray( nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of bytes to read

Returns :

a sequence of bytes with receive buffer contents

On failure, throws an exception or returns a negative error code.

serialport→readBin()serialport.readBin() YSerialPort

Reads data from the receive buffer as a binary buffer, starting at current stream position.

```
function readBin( nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of bytes to read

Returns :

a binary object with receive buffer contents

On failure, throws an exception or returns a negative error code.

serialport→readByte()serialport.readByte() YSerialPort

Reads one byte from the receive buffer, starting at current stream position.

function **readByte**( )

If data at current stream position is not available anymore in the receive buffer, or if there is no data available yet, the function returns YAPI_NO_MORE_DATA.

Returns :
the next byte

On failure, throws an exception or returns a negative error code.

serialport→readHex()serialport.readHex() YSerialPort

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

```
function readHex( nBytes)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nBytes the maximum number of bytes to read

Returns :

a string with receive buffer contents, encoded in hexadecimal

On failure, throws an exception or returns a negative error code.

serialport→readLine()serialport.readLine() YSerialPort

Reads a single line (or message) from the receive buffer, starting at current stream position.

function **readLine**( )

This function is intended to be used when the serial port is configured for a message protocol, such as 'Line' mode or frame protocols.

If data at current stream position is not available anymore in the receive buffer, the function returns the oldest available line and moves the stream position just after. If no new full line is received, the function returns an empty line.

Returns :
a string with a single line of text

On failure, throws an exception or returns a negative error code.

serialport→readMessages() YSerialPort
serialport.readMessages()

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

function **readMessages**(**pattern**, **maxWait**)

This function will only compare and return printable characters in the message strings. Binary protocols are handled as hexadecimal strings.

The search returns all messages matching the expression provided as argument in the buffer. If no matching message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

Parameters :

pattern a limited regular expression describing the expected message format, or an empty string if all messages should be returned (no filtering). When using binary protocols, the format applies to the hexadecimal representation of the message.

maxWait the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

Returns :

an array of strings containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.

serialport→readStr()serialport.readStr() YSerialPort

Reads data from the receive buffer as a string, starting at current stream position.

```
function readStr( nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of characters to read

Returns :

a string with receive buffer contents

On failure, throws an exception or returns a negative error code.

serialport→read_avail()serialport.read_avail() YSerialPort

Returns the number of bytes available to read in the input buffer starting from the current absolute stream position pointer of the API object.

function **read_avail**( )

Returns :
the number of bytes available to read

serialport→read_seek()serialport.read_seek() YSerialPort

Changes the current internal stream position to the specified value.

function **read_seek**(**absPos**)

This function does not affect the device, it only changes the value stored in the API object for the next read operations.

Parameters :

absPos the absolute position index for next read operations.

Returns :

nothing.

serialport→read_tell()serialport.read_tell() YSerialPort

Returns the current absolute stream position pointer of the API object.

function **read_tell**( )

Returns :
the absolute position index for next read operations.

serialport→registerValueCallback() YSerialPort
serialport.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.

serialport→reset()serialport.reset() YSerialPort

Clears the serial port buffer and resets counters to zero.

function **reset**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→selectJob()serialport.selectJob() YSerialPort

Load and start processing the specified job file.

```
function selectJob( jobfile)
```

The file must have been previously created using the user interface or uploaded on the device filesystem using the `uploadJob()` function.

Parameters :

jobfile name of the job file (on the device filesystem)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_RTS() YSerialPort
serialport→setRTS()serialport.set_RTS()

Manually sets the state of the RTS line.

```
function set_RTS( val)
```

This function has no effect when hardware handshake is enabled, as the RTS line is driven automatically.

Parameters :

val 1 to turn RTS on, 0 to turn RTS off

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_currentJob() YSerialPort
serialport→setCurrentJob()
serialport.set_currentJob()

Changes the job to use when the device is powered on.

function **set_currentJob**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the job to use when the device is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_logicalName() YSerialPort
serialport→setLogicalName()
serialport.set_logicalName()

Changes the logical name of the serial port.

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 serial port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_protocol() YSerialPort
serialport→setProtocol()serialport.set_protocol()

Changes the type of protocol used over the serial line.

function **set_protocol**(**newval**)

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Wiegand-ASCII" for Wiegand messages in ASCII mode, "Wiegand-26","Wiegand-34", etc for Wiegand messages in byte mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream. The suffix "/[wait]ms" can be added to reduce the transmit rate so that there is always at lest the specified number of milliseconds between each bytes sent.

Parameters :

newval a string corresponding to the type of protocol used over the serial line

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_serialMode() YSerialPort
serialport→setSerialMode()
serialport.set_serialMode()

Changes the serial port communication parameters, with a string such as "9600,8N1".

function **set_serialMode**(**newval**)

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix can be added to enable flow control: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Parameters :

newval a string corresponding to the serial port communication parameters, with a string such as "9600,8N1"

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_startupJob() YSerialPort
serialport→setStartupJob()
serialport.set_startupJob()

Changes the job to use when the device is powered on.

function **set_startupJob**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the job to use when the device is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_userData() YSerialPort
serialport→setUserData()serialport.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

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

serialport→set_voltageLevel() YSerialPort
serialport→setVoltageLevel()
serialport.set_voltageLevel()

Changes the voltage type used on the serial line.

function **set_voltageLevel**(**newval**)

Valid values will depend on the Yoctopuce device model featuring the serial port feature. Check your device documentation to find out which values are valid for that specific model. Trying to set an invalid value will have no effect.

Parameters :

newval a value among Y_VOLTAGELEVEL_OFF, Y_VOLTAGELEVEL_TTL3V, Y_VOLTAGELEVEL_TTL3VR, Y_VOLTAGELEVEL_TTL5V, Y_VOLTAGELEVEL_TTL5VR, Y_VOLTAGELEVEL_RS232 and Y_VOLTAGELEVEL_RS485 corresponding to the voltage type used on the serial line

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→unmuteValueCallbacks() YSerialPort
serialport.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→uploadJob()serialport.uploadJob() YSerialPort

Saves the job definition string (JSON data) into a job file.

function **uploadJob**(**jobfile**, **jsonDef**)

The job file can be later enabled using `selectJob()`.

Parameters :

jobfile name of the job file to save on the device filesystem

jsonDef a string containing a JSON definition of the job

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→wait_async()serialport.wait_async() YSerialPort

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

serialport→writeArray()serialport.writeArray() YSerialPort

Sends a byte sequence (provided as a list of bytes) to the serial port.

function **writeArray**(**byteList**)

Parameters :

byteList a list of byte codes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeBin()serialport.writeBin() YSerialPort

Sends a binary buffer to the serial port, as is.

function **writeBin**(**buff**)

Parameters :

buff the binary buffer to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeByte()serialport.writeByte() YSerialPort

Sends a single byte to the serial port.

function **writeByte**(**code**)

Parameters :

code the byte to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeHex()serialport.writeHex() YSerialPort

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

function **writeHex**(**hexString**)

Parameters :

hexString a string of hexadecimal byte codes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeLine()serialport.writeLine() YSerialPort

Sends an ASCII string to the serial port, followed by a line break (CR LF).

function **writeLine**(**text**)

Parameters :

text the text string to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeMODBUS()serialport.writeMODBUS() YSerialPort

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

function **writeMODBUS**(**hexString**)

The message must start with the slave address. The MODBUS CRC/LRC is automatically added by the function. This function does not wait for a reply.

Parameters :

hexString a hexadecimal message string, including device address but no CRC/LRC

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeStr()serialport.writeStr() YSerialPort

Sends an ASCII string to the serial port, as is.

function **writeStr**(**text**)

Parameters :

text the text string to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.61. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_servo.js'></script>
cpp	#include "yocto_servo.h"
m	#import "yocto_servo.h"
pas	uses yocto_servo;
vb	yocto_servo.vb
cs	yocto_servo.cs
java	import com.yoctopuce.YoctoAPI.YServo;
uwp	import com.yoctopuce.YoctoAPI.YServo;
py	from yocto_servo import *
php	require_once('yocto_servo.php');
es	in HTML: <script src="../../lib/yocto_servo.js"></script> in node.js: require('yoctolib-es2017/yocto_servo.js');

Global functions
yFindServo(**func**) Retrieves a servo for a given identifier.
yFindServoInContext(**yctx**, **func**) Retrieves a servo for a given identifier in a YAPI context.
yFirstServo() Starts the enumeration of servos currently accessible.
yFirstServoInContext(**yctx**) Starts the enumeration of servos currently accessible.
YServo methods
servo→clearCache() Invalidates the cache.
servo→describe() Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
servo→get_advertisedValue() Returns the current value of the servo (no more than 6 characters).
servo→get_enabled() Returns the state of the servos.
servo→get_enabledAtPowerOn() Returns the servo signal generator state at power up.
servo→get_errorMessage() Returns the error message of the latest error with the servo.
servo→get_errorType() Returns the numerical error code of the latest error with the servo.
servo→get_friendlyName() Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.
servo→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

servo→get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

servo→get_hardwareId()

Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.

servo→get_logicalName()

Returns the logical name of the servo.

servo→get_module()

Gets the YModule object for the device on which the function is located.

servo→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_positionAtPowerOn()

Returns the servo position at device power up.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

servo→isOnline()

Checks if the servo is currently reachable, without raising any error.

servo→isOnline_async(callback, context)

Checks if the servo is currently reachable, without raising any error (asynchronous version).

servo→load(msValidity)

Preloads the servo cache with a specified validity duration.

servo→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

servo→load_async(msValidity, callback, context)

Preloads the servo cache with a specified validity duration (asynchronous version).

servo→move(target, ms_duration)

Performs a smooth move at constant speed toward a given position.

servo→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

servo→nextServo()

Continues the enumeration of servos started using yFirstServo().

servo→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

servo→set_enabled(newval)

Stops or starts the servo.

servo→set_enabledAtPowerOn(newval)

Configure the servo signal generator state at power up.

servo→set_logicalName(newval)

Changes the logical name of the servo.

servo→set_neutral(newval)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→set_position(newval)

Changes immediately the servo driving position.

servo→set_positionAtPowerOn(newval)

Configure the servo position at device power up.

servo→set_range(newval)

Changes the range of use of the servo, specified in per cents.

servo→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

servo→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

servo→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YServo.FindServo() YServo
yFindServo()yFindServo()

Retrieves a servo for a given identifier.

function **FindServo**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the servo is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YServo.isOnline()` to test if the servo is indeed online at a given time. In case of ambiguity when looking for a servo by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the servo

Returns :

a `YServo` object allowing you to drive the servo.

YServo.FindServoInContext() YServo
yFindServoInContext()

Retrieves a servo for a given identifier in a YAPI context.

function **FindServoInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the servo

Returns :

a `YServo` object allowing you to drive the servo.

YServo.FirstServo() YServo yFirstServo()yFirstServo()

Starts the enumeration of servos currently accessible.

function **FirstServo**( )

Use the method `YServo.nextServo()` to iterate on next servos.

Returns :

a pointer to a `YServo` object, corresponding to the first servo currently online, or a `null` pointer if there are none.

YServo.FirstServoInContext() YServo yFirstServoInContext()

Starts the enumeration of servos currently accessible.

function **FirstServoInContext**(**yctx**)

Use the method `YServo.nextServo()` to iterate on next servos.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YServo` object, corresponding to the first servo currently online, or a `null` pointer if there are none.

servo→clearCache()servo.clearCache() YServo

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the servo attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

servo→describe()servo.describe() YServo

Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the servo (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

servo→get_advertisedValue() YServo
servo→advertisedValue()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_enabled() YServo
servo→enabled()servo.get_enabled()

Returns the state of the servos.

```
function get_enabled( )
```

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the servos

On failure, throws an exception or returns Y_ENABLED_INVALID.

servo→get_enabledAtPowerOn() YServo
servo→enabledAtPowerOn()
servo.get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

function **get_enabledAtPowerOn**( )

Returns :

either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the servo signal generator state at power up

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.

servo→get_errorMessage() YServo
servo→errorMessage()servo.get_errorMessage()

Returns the error message of the latest error with the servo.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the servo object

servo→get_errorType() YServo
servo→errorType()servo.get_errorType()

Returns the numerical error code of the latest error with the servo.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the servo object

servo→get_friendlyName() YServo
servo→friendlyName()servo.get_friendlyName()

Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the servo if they are defined, otherwise the serial number of the module and the hardware identifier of the servo (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the servo using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

servo→get_functionDescriptor() YServo
servo→functionDescriptor()
servo.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

servo→get_functionId() YServo
servo→functionId()servo.get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the servo (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

servo→get_hardwareId() YServo
servo→hardwareId()servo.get_hardwareId()

Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the servo (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the servo (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

servo→get_logicalName() YServo
servo→logicalName()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() YServo
servo→module()servo.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

servo→get_neutral() YServo
servo→neutral()servo.get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

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() YServo
servo→position()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_positionAtPowerOn() YServo
servo→positionAtPowerOn()
servo.get_positionAtPowerOn()

Returns the servo position at device power up.

function **get_positionAtPowerOn**( )

Returns :
an integer corresponding to the servo position at device power up

On failure, throws an exception or returns `Y_POSITIONATPOWERON_INVALID`.

servo→get_range() YServo
servo→range()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() YServo
servo→userData()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()servo.isOnline() YServo

Checks if the servo is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

Returns :

`true` if the servo can be reached, and `false` otherwise

servo→load()servo.load() YServo

Preloads the servo cache with a specified validity duration.

```
function load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()servo.loadAttribute() YServo

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

servo→move()servo.move() YServo

Performs a smooth move at constant speed toward a given position.

function **move**(**target**, **ms_duration**)

Parameters :

target	new position at the end of the move
ms_duration	total duration of the move, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→muteValueCallbacks() YServo
servo.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→nextServo()servo.nextServo() YServo

Continues the enumeration of servos started using `yFirstServo()`.

function **nextServo**( )

Returns :

a pointer to a `YServo` object, corresponding to a servo currently online, or a `null` pointer if there are no more servos to enumerate.

servo→registerValueCallback() YServo
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_enabled() YServo
servo→setEnabled()servo.set_enabled()

Stops or starts the servo.

function **set_enabled**(**newval**)

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_enabledAtPowerOn() YServo
servo→setEnabledAtPowerOn()
servo.set_enabledAtPowerOn()

Configure the servo signal generator state at power up.

function **set_enabledAtPowerOn**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_logicalName() YServo
servo→setLogicalName()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() YServo
servo→setNeutral()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. Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_position() YServo
servo→setPosition()servo.set_position()

Changes immediately the servo driving position.

function **set_position**(**newval**)

Parameters :

newval an integer corresponding to immediately the servo driving position

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_positionAtPowerOn() YServo
servo→setPositionAtPowerOn()
servo.set_positionAtPowerOn()

Configure the servo position at device power up.

function **set_positionAtPowerOn**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_range() YServo
servo→setRange()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. Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer corresponding to the range of use of the servo, specified in per cents

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_userData() YServo
servo→setUserData()servo.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

```
function set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

servo→unmuteValueCallbacks() YServo
servo.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→wait_async()servo.wait_async() YServo

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.62. SPI Port function interface

The SpiPort function interface allows you to fully drive a Yoctopuce SPI port, to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce SPI ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_spiport.js'></script>
cpp	#include "yocto_spiport.h"
m	#import "yocto_spiport.h"
pas	uses yocto_spiport;
vb	yocto_spiport.vb
cs	yocto_spiport.cs
java	import com.yoctopuce.YoctoAPI.YSpiPort;
uwp	import com.yoctopuce.YoctoAPI.YSpiPort;
py	from yocto_spiport import *
php	require_once('yocto_spiport.php');
es	in HTML: <script src="../../lib/yocto_spiport.js"></script> in node.js: require('yoctolib-es2017/yocto_spiport.js');

Global functions
yFindSpiPort(**func**) Retrieves a SPI port for a given identifier.
yFindSpiPortInContext(**yctx**, **func**) Retrieves a SPI port for a given identifier in a YAPI context.
yFirstSpiPort() Starts the enumeration of SPI ports currently accessible.
yFirstSpiPortInContext(**yctx**) Starts the enumeration of SPI ports currently accessible.
`YSpiPort` methods
spiport→clearCache() Invalidates the cache.
spiport→describe() Returns a short text that describes unambiguously the instance of the SPI port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
spiport→get_advertisedValue() Returns the current value of the SPI port (no more than 6 characters).
spiport→get_currentJob() Returns the name of the job file currently in use.
spiport→get_errCount() Returns the total number of communication errors detected since last reset.
spiport→get_errorMessage() Returns the error message of the latest error with the SPI port.
spiport→get_errorType() Returns the numerical error code of the latest error with the SPI port.
spiport→get_friendlyName() Returns a global identifier of the SPI port in the format `MODULE_NAME.FUNCTION_NAME`.

spiport→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

spiport→get_functionId()

Returns the hardware identifier of the SPI port, without reference to the module.

spiport→get_hardwareId()

Returns the unique hardware identifier of the SPI port in the form SERIAL.FUNCTIONID.

spiport→get_lastMsg()

Returns the latest message fully received (for Line and Frame protocols).

spiport→get_logicalName()

Returns the logical name of the SPI port.

spiport→get_module()

Gets the YModule object for the device on which the function is located.

spiport→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

spiport→get_protocol()

Returns the type of protocol used over the serial line, as a string.

spiport→get_rxCount()

Returns the total number of bytes received since last reset.

spiport→get_rxMsgCount()

Returns the total number of messages received since last reset.

spiport→get_shitftSampling()

Returns true when the SDI line phase is shifted with regards to the SDO line.

spiport→get_spiMode()

Returns the SPI port communication parameters, as a string such as "125000,0,msb".

spiport→get_ssPolarity()

Returns the SS line polarity.

spiport→get_startupJob()

Returns the job file to use when the device is powered on.

spiport→get_txCount()

Returns the total number of bytes transmitted since last reset.

spiport→get_txMsgCount()

Returns the total number of messages send since last reset.

spiport→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

spiport→get_voltageLevel()

Returns the voltage level used on the serial line.

spiport→isOnline()

Checks if the SPI port is currently reachable, without raising any error.

spiport→isOnline_async(**callback**, **context**)

Checks if the SPI port is currently reachable, without raising any error (asynchronous version).

spiport→load(**msValidity**)

Preloads the SPI port cache with a specified validity duration.

spiport→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

spiport→load_async(**msValidity**, **callback**, **context**)

Preloads the SPI port cache with a specified validity duration (asynchronous version).

spiport→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

spiport→nextSpiPort()

Continues the enumeration of SPI ports started using `yFirstSpiPort()`.

spiport→queryLine(**query**, **maxWait**)

Sends a text line query to the serial port, and reads the reply, if any.

spiport→readArray(**nChars**)

Reads data from the receive buffer as a list of bytes, starting at current stream position.

spiport→readBin(**nChars**)

Reads data from the receive buffer as a binary buffer, starting at current stream position.

spiport→readByte()

Reads one byte from the receive buffer, starting at current stream position.

spiport→readHex(**nBytes**)

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

spiport→readLine()

Reads a single line (or message) from the receive buffer, starting at current stream position.

spiport→readMessages(**pattern**, **maxWait**)

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

spiport→readStr(**nChars**)

Reads data from the receive buffer as a string, starting at current stream position.

spiport→read_avail()

Returns the number of bytes available to read in the input buffer starting from the current absolute stream position pointer of the API object.

spiport→read_seek(**absPos**)

Changes the current internal stream position to the specified value.

spiport→read_tell()

Returns the current absolute stream position pointer of the API object.

spiport→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

spiport→reset()

Clears the serial port buffer and resets counters to zero.

spiport→selectJob(**jobfile**)

Load and start processing the specified job file.

spiport→set_SS(**val**)

Manually sets the state of the SS line.

spiport→set_currentJob(**newval**)

Changes the job to use when the device is powered on.

spiport→set_logicalName(**newval**)

Changes the logical name of the SPI port.

spiport→set_protocol(**newval**)

Changes the type of protocol used over the serial line.

spiport→set_shitftSampling(**newval**)

Changes the SDI line sampling shift.

spiport→set_spiMode(**newval**)

Changes the SPI port communication parameters, with a string such as "125000,0,msb".

spiport→set_ssPolarity(newval)

Changes the SS line polarity.

spiport→set_startupJob(newval)

Changes the job to use when the device is powered on.

spiport→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

spiport→set_voltageLevel(newval)

Changes the voltage type used on the serial line.

spiport→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

spiport→uploadJob(jobfile, jsonDef)

Saves the job definition string (JSON data) into a job file.

spiport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

spiport→writeArray(byteList)

Sends a byte sequence (provided as a list of bytes) to the serial port.

spiport→writeBin(buff)

Sends a binary buffer to the serial port, as is.

spiport→writeByte(code)

Sends a single byte to the serial port.

spiport→writeHex(hexString)

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

spiport→writeLine(text)

Sends an ASCII string to the serial port, followed by a line break (CR LF).

spiport→writeStr(text)

Sends an ASCII string to the serial port, as is.

YSpiPort.FindSpiPort() YSpiPort yFindSpiPort()yFindSpiPort()

Retrieves a SPI port for a given identifier.

```
function FindSpiPort( 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 SPI port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSpiPort.isOnline()` to test if the SPI port is indeed online at a given time. In case of ambiguity when looking for a SPI 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the SPI port

Returns :

a `YSpiPort` object allowing you to drive the SPI port.

YSpiPort.FindSpiPortInContext() YSpiPort yFindSpiPortInContext()

Retrieves a SPI port for a given identifier in a YAPI context.

function **FindSpiPortInContext**(**yctx**, **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 SPI port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSpiPort.isOnline()` to test if the SPI port is indeed online at a given time. In case of ambiguity when looking for a SPI 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 :

yctx a YAPI context

func a string that uniquely characterizes the SPI port

Returns :

a `YSpiPort` object allowing you to drive the SPI port.

YSpiPort.FirstSpiPort() YSpiPort yFirstSpiPort()yFirstSpiPort()

Starts the enumeration of SPI ports currently accessible.

function **FirstSpiPort**( )

Use the method `YSpiPort.nextSpiPort()` to iterate on next SPI ports.

Returns :

a pointer to a `YSpiPort` object, corresponding to the first SPI port currently online, or a `null` pointer if there are none.

YSpiPort.FirstSpiPortInContext() YSpiPort
yFirstSpiPortInContext()

Starts the enumeration of SPI ports currently accessible.

function **FirstSpiPortInContext**(**yctx**)

Use the method `YSpiPort.nextSpiPort()` to iterate on next SPI ports.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YSpiPort` object, corresponding to the first SPI port currently online, or a `null` pointer if there are none.

spiport→clearCache()spiport.clearCache() YSpiPort

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the SPI port attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

spiport→describe()spiport.describe() YSpiPort

Returns a short text that describes unambiguously the instance of the SPI port in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the SPI port (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

spiport→get_advertisedValue() YSpiPort
spiport→advertisedValue()
spiport.get_advertisedValue()

Returns the current value of the SPI port (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the SPI port (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

spiport→get_currentJob() YSpiPort
spiport→currentJob()spiport.get_currentJob()

Returns the name of the job file currently in use.

function **get_currentJob**( )

Returns :
a string corresponding to the name of the job file currently in use

On failure, throws an exception or returns `Y_CURRENTJOB_INVALID`.

spiport→get_errCount() YSpiPort
spiport→errCount()spiport.get_errCount()

Returns the total number of communication errors detected since last reset.

function **get_errCount**( )

Returns :
an integer corresponding to the total number of communication errors detected since last reset

On failure, throws an exception or returns `Y_ERRCOUNT_INVALID`.

spiport→get_errorMessage() YSpiPort
spiport→errorMessage()spiport.get_errorMessage()

Returns the error message of the latest error with the SPI port.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the SPI port object

spiport→get_errorType() YSpiPort
spiport→errorType()spiport.get_errorType()

Returns the numerical error code of the latest error with the SPI port.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the SPI port object

spiport→get_friendlyName() YSpiPort
spiport→friendlyName()spiport.get_friendlyName()

Returns a global identifier of the SPI port in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the SPI port if they are defined, otherwise the serial number of the module and the hardware identifier of the SPI port (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the SPI port using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

spiport→get_functionDescriptor() YSpiPort
spiport→functionDescriptor()
spiport.get_functionDescriptor()

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.

spiport→get_functionId() YSpiPort
spiport→functionId()spiport.get_functionId()

Returns the hardware identifier of the SPI port, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the SPI port (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

spiport→get_hardwareId() YSpiPort
spiport→hardwareId()spiport.get_hardwareId()

Returns the unique hardware identifier of the SPI port in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the SPI port (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the SPI port (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

spiport→get_lastMsg() YSpiPort
spiport→lastMsg()spiport.get_lastMsg()

Returns the latest message fully received (for Line and Frame protocols).

function **get_lastMsg**( )

Returns :
a string corresponding to the latest message fully received (for Line and Frame protocols)

On failure, throws an exception or returns `Y_LASTMSG_INVALID`.

spiport→get_logicalName() YSpiPort
spiport→logicalName()spiport.get_logicalName()

Returns the logical name of the SPI port.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the SPI port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

spiport→get_module() YSpiPort
spiport→module()spiport.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

spiport→get_protocol() YSpiPort
spiport→protocol()spiport.get_protocol()

Returns the type of protocol used over the serial line, as a string.

function **get_protocol**( )

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

Returns :
a string corresponding to the type of protocol used over the serial line, as a string

On failure, throws an exception or returns `Y_PROTOCOL_INVALID`.

spiport→get_rxCount() YSpiPort
spiport→rxCount()spiport.get_rxCount()

Returns the total number of bytes received since last reset.

function **get_rxCount**( )

Returns :
an integer corresponding to the total number of bytes received since last reset

On failure, throws an exception or returns `Y_RXCOUNT_INVALID`.

spiport→get_rxMsgCount() YSpiPort
spiport→rxMsgCount()spiport.get_rxMsgCount()

Returns the total number of messages received since last reset.

function **get_rxMsgCount**( )

Returns :
an integer corresponding to the total number of messages received since last reset

On failure, throws an exception or returns `Y_RXMSGCOUNT_INVALID`.

spiport→get_shitftSampling() YSpiPort
spiport→shitftSampling()spiport.get_shitftSampling()

Returns true when the SDI line phase is shifted with regards to the SDO line.

function **get_shitftSampling**( )

Returns :

either `Y_SHITFTSAMPLING_OFF` or `Y_SHITFTSAMPLING_ON`, according to true when the SDI line phase is shifted with regards to the SDO line

On failure, throws an exception or returns `Y_SHITFTSAMPLING_INVALID`.

spiport→get_spiMode() YSpiPort
spiport→spiMode()spiport.get_spiMode()

Returns the SPI port communication parameters, as a string such as "125000,0,msb".

function **get_spiMode**( )

The string includes the baud rate, the SPI mode (between 0 and 3) and the bit order.

Returns :
a string corresponding to the SPI port communication parameters, as a string such as "125000,0,msb"

On failure, throws an exception or returns `Y_SPIMODE_INVALID`.

spiport→get_ssPolarity() YSpiPort
spiport→ssPolarity()spiport.get_ssPolarity()

Returns the SS line polarity.

function **get_ssPolarity**( )

Returns :

either Y_SSPOLARITY_ACTIVE_LOW or Y_SSPOLARITY_ACTIVE_HIGH, according to the SS line polarity

On failure, throws an exception or returns Y_SSPOLARITY_INVALID.

spiport→get_startupJob() YSpiPort
spiport→startupJob()spiport.get_startupJob()

Returns the job file to use when the device is powered on.

function **get_startupJob**( )

Returns :
a string corresponding to the job file to use when the device is powered on

On failure, throws an exception or returns Y_STARTUPJOB_INVALID.

spiport→get_txCount() YSpiPort
spiport→txCount()spiport.get_txCount()

Returns the total number of bytes transmitted since last reset.

function **get_txCount**( )

Returns :
an integer corresponding to the total number of bytes transmitted since last reset

On failure, throws an exception or returns `Y_TXCOUNT_INVALID`.

spiport→get_txMsgCount() YSpiPort
spiport→txMsgCount()spiport.get_txMsgCount()

Returns the total number of messages send since last reset.

function **get_txMsgCount**( )

Returns :
an integer corresponding to the total number of messages send since last reset

On failure, throws an exception or returns `Y_TXMSGCOUNT_INVALID`.

spiport→get_userData() YSpiPort
spiport→userData()spiport.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.

spiport→get_voltageLevel() YSpiPort
spiport→voltageLevel()spiport.get_voltageLevel()

Returns the voltage level used on the serial line.

function **get_voltageLevel**( )

Returns :

a value among `Y_VOLTAGELEVEL_OFF`, `Y_VOLTAGELEVEL_TTL3V`, `Y_VOLTAGELEVEL_TTL3VR`, `Y_VOLTAGELEVEL_TTL5V`, `Y_VOLTAGELEVEL_TTL5VR`, `Y_VOLTAGELEVEL_RS232` and `Y_VOLTAGELEVEL_RS485` corresponding to the voltage level used on the serial line

On failure, throws an exception or returns `Y_VOLTAGELEVEL_INVALID`.

spiport→isOnline()spiport.isOnline() YSpiPort

Checks if the SPI port is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the SPI port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the SPI port.

Returns :
`true` if the SPI port can be reached, and `false` otherwise

spiport→load()spiport.load() YSpiPort

Preloads the SPI port cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

spiport→loadAttribute()spiport.loadAttribute() YSpiPort

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

spiport→muteValueCallbacks() YSpiPort
spiport.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→nextSpiPort()spiport.nextSpiPort() YSpiPort

Continues the enumeration of SPI ports started using `yFirstSpiPort()`.

function **nextSpiPort**( )

Returns :

a pointer to a `YSpiPort` object, corresponding to a SPI port currently online, or a `null` pointer if there are no more SPI ports to enumerate.

spiport→queryLine()spiport.queryLine() YSpiPort

Sends a text line query to the serial port, and reads the reply, if any.

function **queryLine**(**query**, **maxWait**)

This function is intended to be used when the serial port is configured for 'Line' protocol.

Parameters :

query the line query to send (without CR/LF)

maxWait the maximum number of milliseconds to wait for a reply.

Returns :

the next text line received after sending the text query, as a string. Additional lines can be obtained by calling readLine or readMessages.

On failure, throws an exception or returns an empty array.

spiport→readArray()spiport.readArray() YSpiPort

Reads data from the receive buffer as a list of bytes, starting at current stream position.

```
function readArray( nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of bytes to read

Returns :

a sequence of bytes with receive buffer contents

On failure, throws an exception or returns a negative error code.

spiport→readBin()spiport.readBin() YSpiPort

Reads data from the receive buffer as a binary buffer, starting at current stream position.

function **readBin**(**nChars**)

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of bytes to read

Returns :

a binary object with receive buffer contents

On failure, throws an exception or returns a negative error code.

spiport→readByte()spiport.readByte() YSpiPort

Reads one byte from the receive buffer, starting at current stream position.

function **readByte**( )

If data at current stream position is not available anymore in the receive buffer, or if there is no data available yet, the function returns YAPI_NO_MORE_DATA.

Returns :
the next byte

On failure, throws an exception or returns a negative error code.

spiport→readHex()spiport.readHex() YSpiPort

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

function **readHex**(**nBytes**)

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nBytes the maximum number of bytes to read

Returns :

a string with receive buffer contents, encoded in hexadecimal

On failure, throws an exception or returns a negative error code.

spiport→readLine()spiport.readLine() YSpiPort

Reads a single line (or message) from the receive buffer, starting at current stream position.

function **readLine**( )

This function is intended to be used when the serial port is configured for a message protocol, such as 'Line' mode or frame protocols.

If data at current stream position is not available anymore in the receive buffer, the function returns the oldest available line and moves the stream position just after. If no new full line is received, the function returns an empty line.

Returns :
a string with a single line of text

On failure, throws an exception or returns a negative error code.

spiport→readMessages()spiport.readMessages() YSpiPort

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

function **readMessages**(**pattern**, **maxWait**)

This function will only compare and return printable characters in the message strings. Binary protocols are handled as hexadecimal strings.

The search returns all messages matching the expression provided as argument in the buffer. If no matching message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

Parameters :

pattern a limited regular expression describing the expected message format, or an empty string if all messages should be returned (no filtering). When using binary protocols, the format applies to the hexadecimal representation of the message.

maxWait the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

Returns :

an array of strings containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.

spiport→readStr()spiport.readStr() YSpiPort

Reads data from the receive buffer as a string, starting at current stream position.

```
function readStr( nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of characters to read

Returns :

a string with receive buffer contents

On failure, throws an exception or returns a negative error code.

spiport→read_avail()spiport.read_avail() YSpiPort

Returns the number of bytes available to read in the input buffer starting from the current absolute stream position pointer of the API object.

function **read_avail**( )

Returns :
the number of bytes available to read

spiport→read_seek()spiport.read_seek() YSpiPort

Changes the current internal stream position to the specified value.

function **read_seek**(**absPos**)

This function does not affect the device, it only changes the value stored in the API object for the next read operations.

Parameters :

absPos the absolute position index for next read operations.

Returns :

nothing.

spiport→read_tell()spiport.read_tell() YSpiPort

Returns the current absolute stream position pointer of the API object.

function **read_tell**( )

Returns :
the absolute position index for next read operations.

spiport→registerValueCallback() YSpiPort
spiport.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.

spiport→reset()spiport.reset() YSpiPort

Clears the serial port buffer and resets counters to zero.

function **reset**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→selectJob()spiport.selectJob() YSpiPort

Load and start processing the specified job file.

function **selectJob**(**jobfile**)

The file must have been previously created using the user interface or uploaded on the device filesystem using the `uploadJob()` function.

Parameters :

jobfile name of the job file (on the device filesystem)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_SS() YSpiPort
spiport→setSS()spiport.set_SS()

Manually sets the state of the SS line.

```
function set_SS( val)
```

This function has no effect when the SS line is handled automatically.

Parameters :

val 1 to turn SS active, 0 to release SS.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_currentJob() YSpiPort
spiport→setCurrentJob()spiport.set_currentJob()

Changes the job to use when the device is powered on.

function **set_currentJob**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the job to use when the device is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_logicalName() YSpiPort
spiport→setLogicalName()spiport.set_logicalName()

Changes the logical name of the SPI port.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the SPI port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_protocol() YSpiPort
spiport→setProtocol()spiport.set_protocol()

Changes the type of protocol used over the serial line.

function **set_protocol**(**newval**)

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream. The suffix "/[wait]ms" can be added to reduce the transmit rate so that there is always at lest the specified number of milliseconds between each bytes sent.

Parameters :

newval a string corresponding to the type of protocol used over the serial line

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_shitftSampling() YSpiPort
spiport→setShitftSampling()
spiport.set_shitftSampling()

Changes the SDI line sampling shift.

function **set_shitftSampling**(**newval**)

When disabled, SDI line is sampled in the middle of data output time. When enabled, SDI line is samples at the end of data output time.

Parameters :

newval either `Y_SHITFTSAMPLING_OFF` or `Y_SHITFTSAMPLING_ON`, according to the SDI line sampling shift

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_spiMode() YSpiPort
spiport→setSpiMode()spiport.set_spiMode()

Changes the SPI port communication parameters, with a string such as "125000,0,msb".

function **set_spiMode**(**newval**)

The string includes the baud rate, the SPI mode (between 0 and 3) and the bit order.

Parameters :

newval a string corresponding to the SPI port communication parameters, with a string such as "125000,0,msb"

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_ssPolarity() YSpiPort
spiport→setSsPolarity()spiport.set_ssPolarity()

Changes the SS line polarity.

function **set_ssPolarity**(**newval**)

Parameters :

newval either `Y_SSPOLARITY_ACTIVE_LOW` or `Y_SSPOLARITY_ACTIVE_HIGH`, according to the SS line polarity

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_startupJob() YSpiPort
spiport→setStartupJob()spiport.set_startupJob()

Changes the job to use when the device is powered on.

function **set_startupJob**(**newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the job to use when the device is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→set_userData() YSpiPort
spiport→setUserData()spiport.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

spiport→set_voltageLevel() YSpiPort
spiport→setVoltageLevel()spiport.set_voltageLevel()

Changes the voltage type used on the serial line.

function **set_voltageLevel**(**newval**)

Valid values will depend on the Yoctopuce device model featuring the serial port feature. Check your device documentation to find out which values are valid for that specific model. Trying to set an invalid value will have no effect.

Parameters :

newval a value among Y_VOLTAGELEVEL_OFF, Y_VOLTAGELEVEL_TTL3V, Y_VOLTAGELEVEL_TTL3VR, Y_VOLTAGELEVEL_TTL5V, Y_VOLTAGELEVEL_TTL5VR, Y_VOLTAGELEVEL_RS232 and Y_VOLTAGELEVEL_RS485 corresponding to the voltage type used on the serial line

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→unmuteValueCallbacks() YSpiPort
spiport.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→uploadJob()spiport.uploadJob() YSpiPort

Saves the job definition string (JSON data) into a job file.

function **uploadJob**(**jobfile**, **jsonDef**)

The job file can be later enabled using `selectJob()`.

Parameters :

jobfile name of the job file to save on the device filesystem

jsonDef a string containing a JSON definition of the job

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→wait_async()spiport.wait_async() YSpiPort

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

spiport→writeArray()spiport.writeArray() YSpiPort

Sends a byte sequence (provided as a list of bytes) to the serial port.

function **writeArray**(**byteList**)

Parameters :

byteList a list of byte codes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→writeBin()spiport.writeBin() YSpiPort

Sends a binary buffer to the serial port, as is.

function **writeBin**(**buff**)

Parameters :

buff the binary buffer to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→writeByte()spiport.writeByte() YSpiPort

Sends a single byte to the serial port.

function **writeByte**(**code**)

Parameters :

code the byte to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→writeHex()spiport.writeHex() YSpiPort

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

function **writeHex**(**hexString**)

Parameters :

hexString a string of hexadecimal byte codes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→writeLine()spiport.writeLine() YSpiPort

Sends an ASCII string to the serial port, followed by a line break (CR LF).

function **writeLine**(**text**)

Parameters :

text the text string to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

spiport→writeStr()spiport.writeStr() YSpiPort

Sends an ASCII string to the serial port, as is.

function **writeStr**(**text**)

Parameters :

text the text string to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.63. StepperMotor function interface

The Yoctopuce application programming interface allows you to drive a stepper motor.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_steppermotor.js'></script>
cpp	#include "yocto_steppermotor.h"
m	#import "yocto_steppermotor.h"
pas	uses yocto_steppermotor;
vb	yocto_steppermotor.vb
cs	yocto_steppermotor.cs
java	import com.yoctopuce.YoctoAPI.YStepperMotor;
uwp	import com.yoctopuce.YoctoAPI.YStepperMotor;
py	from yocto_steppermotor import *
php	require_once('yocto_steppermotor.php');
es	in HTML: <script src="../../lib/yocto_steppermotor.js"></script> in node.js: require('yoctolib-es2017/yocto_steppermotor.js');

Global functions

yFindStepperMotor(**func**)

Retrieves a stepper motor for a given identifier.

yFindStepperMotorInContext(**yctx**, **func**)

Retrieves a stepper motor for a given identifier in a YAPI context.

yFirstStepperMotor()

Starts the enumeration of stepper motors currently accessible.

yFirstStepperMotorInContext(**yctx**)

Starts the enumeration of stepper motors currently accessible.

`YStepperMotor` methods

steppermotor→abortAndBrake()

Stops the motor smoothly as soon as possible, without waiting for ongoing move completion.

steppermotor→abortAndHiZ()

Turn the controller into Hi-Z mode immediately, without waiting for ongoing move completion.

steppermotor→alertStepOut()

Move one step in the direction opposite the direction set when the most recent alert was raised.

steppermotor→changeSpeed(**speed**)

Starts the motor at a given speed.

steppermotor→clearCache()

Invalidates the cache.

steppermotor→describe()

Returns a short text that describes unambiguously the instance of the stepper motor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

steppermotor→emergencyStop()

Stops the motor with an emergency alert, without taking any additional precaution.

steppermotor→findHomePosition(**speed**)

Starts the motor backward at the specified speed, to search for the motor home position.

steppermotor→get_advertisedValue()

Returns the current value of the stepper motor (no more than 6 characters).

steppermotor→get_auxSignal()

Returns the current value of the signal generated on the auxiliary output.

steppermotor→get_diags()
Returns the stepper motor controller diagnostics, as a bitmap.

steppermotor→get_errorMessage()
Returns the error message of the latest error with the stepper motor.

steppermotor→get_errorType()
Returns the numerical error code of the latest error with the stepper motor.

steppermotor→get_friendlyName()
Returns a global identifier of the stepper motor in the format `MODULE_NAME.FUNCTION_NAME`.

steppermotor→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

steppermotor→get_functionId()
Returns the hardware identifier of the stepper motor, without reference to the module.

steppermotor→get_hardwareId()
Returns the unique hardware identifier of the stepper motor in the form `SERIAL.FUNCTIONID`.

steppermotor→get_logicalName()
Returns the logical name of the stepper motor.

steppermotor→get_maxAccel()
Returns the maximal motor acceleration, measured in steps per second^2.

steppermotor→get_maxSpeed()
Returns the maximal motor speed, measured in steps per second.

steppermotor→get_module()
Gets the `YModule` object for the device on which the function is located.

steppermotor→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

steppermotor→get_motorState()
Returns the motor working state.

steppermotor→get_overcurrent()
Returns the overcurrent alert and emergency stop threshold, measured in mA.

steppermotor→get_pullinSpeed()
Returns the motor speed immediately reachable from stop state, measured in steps per second.

steppermotor→get_speed()
Returns current motor speed, measured in steps per second.

steppermotor→get_stepPos()
Returns the current logical motor position, measured in steps.

steppermotor→get_stepping()
Returns the stepping mode used to drive the motor.

steppermotor→get_tCurrRun()
Returns the torque regulation current when the motor is running, measured in mA.

steppermotor→get_tCurrStop()
Returns the torque regulation current when the motor is stopped, measured in mA.

steppermotor→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

steppermotor→isOnline()
Checks if the stepper motor is currently reachable, without raising any error.

steppermotor→isOnline_async(callback, context)

Checks if the stepper motor is currently reachable, without raising any error (asynchronous version).

steppermotor→load(msValidity)

Preloads the stepper motor cache with a specified validity duration.

steppermotor→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

steppermotor→load_async(msValidity, callback, context)

Preloads the stepper motor cache with a specified validity duration (asynchronous version).

steppermotor→moveRel(relPos)

Starts the motor to reach a given relative position.

steppermotor→moveTo(absPos)

Starts the motor to reach a given absolute position.

steppermotor→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

steppermotor→nextStepperMotor()

Continues the enumeration of stepper motors started using `yFirstStepperMotor()`.

steppermotor→pause(waitMs)

Keep the motor in the same state for the specified amount of time, before processing next command.

steppermotor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

steppermotor→reset()

Reinitialize the controller and clear all alert flags.

steppermotor→set_auxSignal(newval)

Changes the value of the signal generated on the auxiliary output.

steppermotor→set_logicalName(newval)

Changes the logical name of the stepper motor.

steppermotor→set_maxAccel(newval)

Changes the maximal motor acceleration, measured in steps per second^2.

steppermotor→set_maxSpeed(newval)

Changes the maximal motor speed, measured in steps per second.

steppermotor→set_overcurrent(newval)

Changes the overcurrent alert and emergency stop threshold, measured in mA.

steppermotor→set_pullinSpeed(newval)

Changes the motor speed immediately reachable from stop state, measured in steps per second.

steppermotor→set_stepPos(newval)

Changes the current logical motor position, measured in steps.

steppermotor→set_stepping(newval)

Changes the stepping mode used to drive the motor.

steppermotor→set_tCurrRun(newval)

Changes the torque regulation current when the motor is running, measured in mA.

steppermotor→set_tCurrStop(newval)

Changes the torque regulation current when the motor is stopped, measured in mA.

steppermotor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

steppermotor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

steppermotor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YStepperMotor.FindStepperMotor() YStepperMotor yFindStepperMotor()yFindStepperMotor()

Retrieves a stepper motor for a given identifier.

function **FindStepperMotor**(**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 stepper motor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YStepperMotor.isOnline()` to test if the stepper motor is indeed online at a given time. In case of ambiguity when looking for a stepper motor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the stepper motor

Returns :

a `YStepperMotor` object allowing you to drive the stepper motor.

YStepperMotor.FindStepperMotorInContext() YStepperMotor yFindStepperMotorInContext()

Retrieves a stepper motor for a given identifier in a YAPI context.

function **FindStepperMotorInContext**(**yctx**, **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 stepper motor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YStepperMotor.isOnline()` to test if the stepper motor is indeed online at a given time. In case of ambiguity when looking for a stepper motor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the stepper motor

Returns :

a `YStepperMotor` object allowing you to drive the stepper motor.

YStepperMotor.FirstStepperMotor() YStepperMotor yFirstStepperMotor()yFirstStepperMotor()

Starts the enumeration of stepper motors currently accessible.

function **FirstStepperMotor**( )

Use the method `YStepperMotor.nextStepperMotor()` to iterate on next stepper motors.

Returns :

a pointer to a `YStepperMotor` object, corresponding to the first stepper motor currently online, or a `null` pointer if there are none.

YStepperMotor.FirstStepperMotorInContext() YStepperMotor yFirstStepperMotorInContext()

Starts the enumeration of stepper motors currently accessible.

function **FirstStepperMotorInContext**(**yctx**)

Use the method `YStepperMotor.nextStepperMotor()` to iterate on next stepper motors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YStepperMotor` object, corresponding to the first stepper motor currently online, or a `null` pointer if there are none.

steppermotor→abortAndBrake() YStepperMotor
steppermotor.abortAndBrake()

Stops the motor smoothly as soon as possible, without waiting for ongoing move completion.

function **abortAndBrake**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→abortAndHiZ() YStepperMotor
steppermotor.abortAndHiZ()

Turn the controller into Hi-Z mode immediately, without waiting for ongoing move completion.

function **abortAndHiZ**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→alertStepOut() YStepperMotor
steppermotor.alertStepOut()

Move one step in the direction opposite the direction set when the most recent alert was raised.

function **alertStepOut**( )

The move occures even if the system is still in alert mode (end switch depressed). Caution. use this function with great care as it may cause mechanical damages !

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→changeSpeed() YStepperMotor
steppermotor.changeSpeed()

Starts the motor at a given speed.

```
function changeSpeed( speed)
```

The time needed to reach the requested speed will depend on the acceleration parameters configured for the motor.

Parameters :

speed desired speed, in steps per second. The minimal non-zero speed is 0.001 pulse per second.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→clearCache() YStepperMotor
steppermotor.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the stepper motor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

steppermotor→describe()steppermotor.describe() YStepperMotor

Returns a short text that describes unambiguously the instance of the stepper motor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the stepper motor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

steppermotor→emergencyStop() YStepperMotor
steppermotor.emergencyStop()

Stops the motor with an emergency alert, without taking any additional precaution.

function **emergencyStop**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→findHomePosition() YStepperMotor
steppermotor.findHomePosition()

Starts the motor backward at the specified speed, to search for the motor home position.

function **findHomePosition**(**speed**)

Parameters :

speed desired speed, in steps per second.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→get_advertisedValue() YStepperMotor
steppermotor→advertisedValue()
steppermotor.get_advertisedValue()

Returns the current value of the stepper motor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the stepper motor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

steppermotor→get_auxSignal() YStepperMotor
steppermotor→auxSignal()
steppermotor.get_auxSignal()

Returns the current value of the signal generated on the auxiliary output.

function **get_auxSignal**( )

Returns :
an integer corresponding to the current value of the signal generated on the auxiliary output

On failure, throws an exception or returns `Y_AUXSIGNAL_INVALID`.

steppermotor→get_diags() YStepperMotor
steppermotor→diags()steppermotor.get_diags()

Returns the stepper motor controller diagnostics, as a bitmap.

function **get_diags**( )

Returns :
an integer corresponding to the stepper motor controller diagnostics, as a bitmap

On failure, throws an exception or returns `Y_DIAGS_INVALID`.

steppermotor→get_errorMessage() YStepperMotor
steppermotor→errorMessage()
steppermotor.get_errorMessage()

Returns the error message of the latest error with the stepper motor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the stepper motor object

steppermotor→get_errorType() YStepperMotor
steppermotor→errorType()
steppermotor.get_errorType()

Returns the numerical error code of the latest error with the stepper motor.

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 the stepper motor object

steppermotor→get_friendlyName() YStepperMotor
steppermotor→friendlyName()
steppermotor.get_friendlyName()

Returns a global identifier of the stepper motor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the stepper motor if they are defined, otherwise the serial number of the module and the hardware identifier of the stepper motor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the stepper motor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

steppermotor→get_functionDescriptor() YStepperMotor
steppermotor→functionDescriptor()
steppermotor.get_functionDescriptor()

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.

steppermotor→get_functionId() YStepperMotor
steppermotor→functionId()
steppermotor.get_functionId()

Returns the hardware identifier of the stepper motor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the stepper motor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

steppermotor→get_hardwareId() YStepperMotor
steppermotor→hardwareId()
steppermotor.get_hardwareId()

Returns the unique hardware identifier of the stepper motor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the stepper motor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the stepper motor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

steppermotor→get_logicalName() YStepperMotor
steppermotor→logicalName()
steppermotor.get_logicalName()

Returns the logical name of the stepper motor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the stepper motor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

steppermotor→get_maxAccel() YStepperMotor
steppermotor→maxAccel()
steppermotor.get_maxAccel()

Returns the maximal motor acceleration, measured in steps per second^2.

function **get_maxAccel**( )

Returns :
a floating point number corresponding to the maximal motor acceleration, measured in steps per second^2

On failure, throws an exception or returns `Y_MAXACCEL_INVALID`.

steppermotor→get_maxSpeed() YStepperMotor
steppermotor→maxSpeed()
steppermotor.get_maxSpeed()

Returns the maximal motor speed, measured in steps per second.

function **get_maxSpeed**( )

Returns :
a floating point number corresponding to the maximal motor speed, measured in steps per second

On failure, throws an exception or returns `Y_MAXSPEED_INVALID`.

steppermotor→get_module() YStepperMotor
steppermotor→module()steppermotor.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

steppermotor→get_motorState() YStepperMotor
steppermotor→motorState()
steppermotor.get_motorState()

Returns the motor working state.

function **get_motorState**( )

Returns :

a value among `Y_MOTORSTATE_ABSENT`, `Y_MOTORSTATE_ALERT`, `Y_MOTORSTATE_HI_Z`, `Y_MOTORSTATE_STOP`, `Y_MOTORSTATE_RUN` and `Y_MOTORSTATE_BATCH` corresponding to the motor working state

On failure, throws an exception or returns `Y_MOTORSTATE_INVALID`.

steppermotor→get_overcurrent() YStepperMotor
steppermotor→overcurrent()
steppermotor.get_overcurrent()

Returns the overcurrent alert and emergency stop threshold, measured in mA.

function **get_overcurrent**( )

Returns :
an integer corresponding to the overcurrent alert and emergency stop threshold, measured in mA

On failure, throws an exception or returns `Y_OVERCURRENT_INVALID`.

steppermotor→get_pullinSpeed() YStepperMotor
steppermotor→pullinSpeed()
steppermotor.get_pullinSpeed()

Returns the motor speed immediately reachable from stop state, measured in steps per second.

function **get_pullinSpeed**( )

Returns :

a floating point number corresponding to the motor speed immediately reachable from stop state, measured in steps per second

On failure, throws an exception or returns `Y_PULLINSPEED_INVALID`.

steppermotor→get_speed() YStepperMotor
steppermotor→speed()steppermotor.get_speed()

Returns current motor speed, measured in steps per second.

function **get_speed**( )

To change speed, use method `changeSpeed()`.

Returns :
a floating point number corresponding to current motor speed, measured in steps per second

On failure, throws an exception or returns `Y_SPEED_INVALID`.

steppermotor→get_stepPos() YStepperMotor
steppermotor→stepPos()steppermotor.get_stepPos()

Returns the current logical motor position, measured in steps.

function **get_stepPos**( )

The value may include a fractional part when micro-stepping is in use.

Returns :
a floating point number corresponding to the current logical motor position, measured in steps

On failure, throws an exception or returns `Y_STEPPOS_INVALID`.

steppermotor→get_stepping() YStepperMotor
steppermotor→stepping()
steppermotor.get_stepping()

Returns the stepping mode used to drive the motor.

function **get_stepping**( )

Returns :

a value among `Y_STEPPING_MICROSTEP16`, `Y_STEPPING_MICROSTEP8`, `Y_STEPPING_MICROSTEP4`, `Y_STEPPING_HALFSTEP` and `Y_STEPPING_FULLSTEP` corresponding to the stepping mode used to drive the motor

On failure, throws an exception or returns `Y_STEPPING_INVALID`.

steppermotor→get_tCurrRun() YStepperMotor
steppermotor→tCurrRun()
steppermotor.get_tCurrRun()

Returns the torque regulation current when the motor is running, measured in mA.

function **get_tCurrRun**( )

Returns :
an integer corresponding to the torque regulation current when the motor is running, measured in mA

On failure, throws an exception or returns `Y_TCURRRUN_INVALID`.

steppermotor→get_tCurrStop() YStepperMotor
steppermotor→tCurrStop()
steppermotor.get_tCurrStop()

Returns the torque regulation current when the motor is stopped, measured in mA.

function **get_tCurrStop**( )

Returns :
an integer corresponding to the torque regulation current when the motor is stopped, measured in mA

On failure, throws an exception or returns `Y_TCURRSTOP_INVALID`.

steppermotor→get_userData() YStepperMotor
steppermotor→userData()
steppermotor.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.

steppermotor→isOnline()steppermotor.isOnline() YStepperMotor

Checks if the stepper motor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the stepper motor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the stepper motor.

Returns :

`true` if the stepper motor can be reached, and `false` otherwise

steppermotor→load()steppermotor.load() YStepperMotor

Preloads the stepper motor 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.

steppermotor→loadAttribute() YStepperMotor
steppermotor.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

steppermotor→moveRel()steppermotor.moveRel() YStepperMotor

Starts the motor to reach a given relative position.

function **moveRel**(**relPos**)

The time needed to reach the requested position will depend on the acceleration and max speed parameters configured for the motor.

Parameters :

relPos relative position, measured in steps from the current position.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→moveTo()steppermotor.moveTo() YStepperMotor

Starts the motor to reach a given absolute position.

function **moveTo**(**absPos**)

The time needed to reach the requested position will depend on the acceleration and max speed parameters configured for the motor.

Parameters :

absPos absolute position, measured in steps from the origin.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→muteValueCallbacks() YStepperMotor
steppermotor.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→nextStepperMotor() YStepperMotor
steppermotor.nextStepperMotor()

Continues the enumeration of stepper motors started using `yFirstStepperMotor()`.

function **nextStepperMotor**( )

Returns :

a pointer to a `YStepperMotor` object, corresponding to a stepper motor currently online, or a `null` pointer if there are no more stepper motors to enumerate.

steppermotor→pause()steppermotor.pause() YStepperMotor

Keep the motor in the same state for the specified amount of time, before processing next command.

function **pause**(**waitMs**)

Parameters :

waitMs wait time, specified in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→registerValueCallback() YStepperMotor
steppermotor.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.

steppermotor→reset()steppermotor.reset() YStepperMotor

Reinitialize the controller and clear all alert flags.

function **reset**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

steppermotor→set_auxSignal() YStepperMotor
steppermotor→setAuxSignal()
steppermotor.set_auxSignal()

Changes the value of the signal generated on the auxiliary output.

function **set_auxSignal**(**newval**)

Acceptable values depend on the auxiliary output signal type configured.

Parameters :

newval an integer corresponding to the value of the signal generated on the auxiliary output

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_logicalName() YStepperMotor
steppermotor→setLogicalName()
steppermotor.set_logicalName()

Changes the logical name of the stepper motor.

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 stepper motor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_maxAccel() YStepperMotor
steppermotor→setMaxAccel()
steppermotor.set_maxAccel()

Changes the maximal motor acceleration, measured in steps per second^2.

function **set_maxAccel**(**newval**)

Parameters :

newval a floating point number corresponding to the maximal motor acceleration, measured in steps per second^2

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_maxSpeed() YStepperMotor
steppermotor→setMaxSpeed()
steppermotor.set_maxSpeed()

Changes the maximal motor speed, measured in steps per second.

function **set_maxSpeed**(**newval**)

Parameters :

newval a floating point number corresponding to the maximal motor speed, measured in steps per second

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_overcurrent() YStepperMotor
steppermotor→setOvercurrent()
steppermotor.set_overcurrent()

Changes the overcurrent alert and emergency stop threshold, measured in mA.

function **set_overcurrent**(**newval**)

Parameters :

newval an integer corresponding to the overcurrent alert and emergency stop threshold, measured in mA

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_pullinSpeed() YStepperMotor
steppermotor→setPullinSpeed()
steppermotor.set_pullinSpeed()

Changes the motor speed immediately reachable from stop state, measured in steps per second.

function **set_pullinSpeed**(**newval**)

Parameters :

newval a floating point number corresponding to the motor speed immediately reachable from stop state, measured in steps per second

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_stepPos() YStepperMotor
steppermotor→setStepPos()
steppermotor.set_stepPos()

Changes the current logical motor position, measured in steps.

function **set_stepPos**(**newval**)

This command does not cause any motor move, as its purpose is only to setup the origin of the position counter. The fractional part of the position, that corresponds to the physical position of the rotor, is not changed. To trigger a motor move, use methods `moveTo()` or `moveRel()` instead.

Parameters :

newval a floating point number corresponding to the current logical motor position, measured in steps

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_stepping() YStepperMotor
steppermotor→setStepping()
steppermotor.set_stepping()

Changes the stepping mode used to drive the motor.

function **set_stepping**(**newval**)

Parameters :

newval a value among `Y_STEPPING_MICROSTEP16`, `Y_STEPPING_MICROSTEP8`, `Y_STEPPING_MICROSTEP4`, `Y_STEPPING_HALFSTEP` and `Y_STEPPING_FULLSTEP` corresponding to the stepping mode used to drive the motor

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_tCurrRun() YStepperMotor
steppermotor→setTCurrRun()
steppermotor.set_tCurrRun()

Changes the torque regulation current when the motor is running, measured in mA.

function **set_tCurrRun**(**newval**)

Parameters :

newval an integer corresponding to the torque regulation current when the motor is running, measured in mA

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_tCurrStop() YStepperMotor
steppermotor→setTCurrStop()
steppermotor.set_tCurrStop()

Changes the torque regulation current when the motor is stopped, measured in mA.

function **set_tCurrStop**(**newval**)

Parameters :

newval an integer corresponding to the torque regulation current when the motor is stopped, measured in mA

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→set_userData() YStepperMotor
steppermotor→setUserData()
steppermotor.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

steppermotor→unmuteValueCallbacks() YStepperMotor
steppermotor.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

steppermotor→wait_async() YStepperMotor
steppermotor.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.64. Temperature function interface

The Yoctopuce class YTemperature allows you to read and configure Yoctopuce temperature sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to configure some specific parameters for some sensors (connection type, temperature mapping table).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_temperature.js'></script>
cpp	#include "yocto_temperature.h"
m	#import "yocto_temperature.h"
pas	uses yocto_temperature;
vb	yocto_temperature.vb
cs	yocto_temperature.cs
java	import com.yoctopuce.YoctoAPI.YTemperature;
uwp	import com.yoctopuce.YoctoAPI.YTemperature;
py	from yocto_temperature import *
php	require_once('yocto_temperature.php');
es	in HTML: <script src="../../lib/yocto_temperature.js"></script> in node.js: require('yoctolib-es2017/yocto_temperature.js');

Global functions
yFindTemperature(**func**) Retrieves a temperature sensor for a given identifier.
yFindTemperatureInContext(**yctx**, **func**) Retrieves a temperature sensor for a given identifier in a YAPI context.
yFirstTemperature() Starts the enumeration of temperature sensors currently accessible.
yFirstTemperatureInContext(**yctx**) 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→clearCache() Invalidates the cache.
temperature→describe() Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
temperature→get_advertisedValue() Returns the current value of the temperature sensor (no more than 6 characters).
temperature→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number.
temperature→get_currentValue() Returns the current value of the temperature, in Celsius, as a floating point number.
temperature→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
temperature→get_errorMessage() Returns the error message of the latest error with the temperature sensor.

temperature→get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

temperature→get_friendlyName()

Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.

temperature→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

temperature→get_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

temperature→get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

temperature→get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

temperature→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

temperature→get_logicalName()

Returns the logical name of the temperature sensor.

temperature→get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

temperature→get_module()

Gets the `YModule` object for the device on which the function is located.

temperature→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

temperature→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

temperature→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

temperature→get_resolution()

Returns the resolution of the measured values.

temperature→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

temperature→get_sensorType()

Returns the temperature sensor type.

temperature→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

temperature→get_signalValue()

Returns the current value of the electrical signal measured by the sensor.

temperature→get_unit()

Returns the measuring unit for the temperature.

temperature→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

temperature→isOnline()

Checks if the temperature sensor is currently reachable, without raising any error.

temperature→isOnline_async(**callback**, **context**)

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

temperature→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

temperature→load(**msValidity**)

Preloads the temperature sensor cache with a specified validity duration.

temperature→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

temperature→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

temperature→loadThermistorResponseTable(**tempValues**, **resValues**)

Retrieves the thermistor response table previously configured using the `set_thermistorResponseTable` function.

temperature→load_async(**msValidity**, **callback**, **context**)

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

temperature→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

temperature→nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

temperature→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

temperature→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

temperature→set_highestValue(**newval**)

Changes the recorded maximal value observed.

temperature→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

temperature→set_logicalName(**newval**)

Changes the logical name of the temperature sensor.

temperature→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

temperature→set_ntcParameters(**res25**, **beta**)

Configures NTC thermistor parameters in order to properly compute the temperature from the measured resistance.

temperature→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

temperature→set_resolution(**newval**)

Changes the resolution of the measured physical values.

temperature→set_sensorType(**newval**)

Modifies the temperature sensor type.

temperature→set_thermistorResponseTable(**tempValues**, **resValues**)

Records a thermistor response table, in order to interpolate the temperature from the measured resistance.

temperature→set_unit(**newval**)

Changes the measuring unit for the measured temperature.

temperature→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

temperature→startDataLogger()

Starts the data logger on the device.

temperature→stopDataLogger()

Stops the datalogger on the device.

temperature→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

temperature→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTemperature.FindTemperature() YTemperature yFindTemperature()yFindTemperature()

Retrieves a temperature sensor for a given identifier.

```
function FindTemperature( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the temperature sensor

Returns :

a `YTemperature` object allowing you to drive the temperature sensor.

YTemperature.FindTemperatureInContext() YTemperature yFindTemperatureInContext()

Retrieves a temperature sensor for a given identifier in a YAPI context.

function **FindTemperatureInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the temperature sensor

Returns :

a `YTemperature` object allowing you to drive the temperature sensor.

YTemperature.FirstTemperature() YTemperature yFirstTemperature()yFirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

function **FirstTemperature**( )

Use the method `YTemperature.nextTemperature()` to iterate on next temperature sensors.

Returns :

a pointer to a `YTemperature` object, corresponding to the first temperature sensor currently online, or a `null` pointer if there are none.

YTemperature.FirstTemperatureInContext() YTemperature yFirstTemperatureInContext()

Starts the enumeration of temperature sensors currently accessible.

function **FirstTemperatureInContext**(**yctx**)

Use the method `YTemperature.nextTemperature()` to iterate on next temperature sensors.

Parameters :

yctx a YAPI context.

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() YTemperature
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 linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→clearCache()temperature.clearCache() YTemperature

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the temperature sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

temperature→describe()temperature.describe() YTemperature

Returns a short text that describes unambiguously the instance of the temperature sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the temperature sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

temperature→get_advertisedValue() YTemperature
temperature→advertisedValue()
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() YTemperature
temperature→currentRawValue()
temperature.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

temperature→get_currentValue() YTemperature
temperature→currentValue()
temperature.get_currentValue()

Returns the current value of the temperature, in Celsius, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the temperature, in Celsius, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

temperature→get_dataLogger() YTemperature
temperature→dataLogger()
temperature.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

temperature→get_errorMessage() YTemperature
temperature→errorMessage()
temperature.get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the temperature sensor object

temperature→get_errorType() YTemperature
temperature→errorType()
temperature.get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the temperature sensor object

temperature→get_friendlyName() YTemperature
temperature→friendlyName()
temperature.get_friendlyName()

Returns a global identifier of the temperature sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the temperature sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the temperature sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the temperature sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

temperature→get_functionDescriptor() YTemperature
temperature→functionDescriptor()
temperature.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

temperature→get_functionId() YTemperature
temperature→functionId()
temperature.get_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the temperature sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

temperature→get_hardwareId() YTemperature
temperature→hardwareId()
temperature.get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the temperature sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the temperature sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

temperature→get_highestValue() YTemperature
temperature→highestValue()
temperature.get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

temperature→get_logFrequency() YTemperature
temperature→logFrequency()
temperature.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

temperature→get_logicalName() YTemperature
temperature→logicalName()
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() YTemperature
temperature→lowestValue()
temperature.get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

temperature→get_module() YTemperature
temperature→module()temperature.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

temperature→get_recordedData() YTemperature
temperature→recordedData()
temperature.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

temperature→get_reportFrequency() YTemperature
temperature→reportFrequency()
temperature.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

temperature→get_resolution() YTemperature
temperature→resolution()
temperature.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

temperature→get_sensorState() YTemperature
temperature→sensorState()
temperature.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :
an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

temperature→get_sensorType() YTemperature
temperature→sensorType()
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, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES, Y_SENSORTYPE_PT100_2WIRES, Y_SENSORTYPE_RES_OHM, Y_SENSORTYPE_RES_NTC, Y_SENSORTYPE_RES_LINEAR and Y_SENSORTYPE_RES_INTERNAL corresponding to the temperature sensor type

On failure, throws an exception or returns Y_SENSORTYPE_INVALID.

temperature→get_signalUnit() YTemperature
temperature→signalUnit()
temperature.get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

function **get_signalUnit**( )

Returns :
a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns `Y_SIGNALUNIT_INVALID`.

temperature→get_signalValue() YTemperature
temperature→signalValue()
temperature.get_signalValue()

Returns the current value of the electrical signal measured by the sensor.

function **get_signalValue**( )

Returns :
a floating point number corresponding to the current value of the electrical signal measured by the sensor

On failure, throws an exception or returns `Y_SIGNALVALUE_INVALID`.

temperature→get_unit() YTemperature
temperature→unit()temperature.get_unit()

Returns the measuring unit for the temperature.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the temperature

On failure, throws an exception or returns `Y_UNIT_INVALID`.

temperature→get_userData() YTemperature
temperature→userData()temperature.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

temperature→isOnline()temperature.isOnline() YTemperature

Checks if the temperature sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

Returns :
`true` if the temperature sensor can be reached, and `false` otherwise

temperature→load()temperature.load() YTemperature

Preloads the temperature sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute() YTemperature
temperature.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

temperature→loadCalibrationPoints() YTemperature
temperature.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→loadThermistorResponseTable() YTemperature
temperature.loadThermistorResponseTable()

Retrieves the thermistor response table previously configured using the `set_thermistorResponseTable` function.

function **loadThermistorResponseTable**(**tempValues**, **resValues**)

This function can only be used with a temperature sensor based on thermistors.

Parameters :

tempValues array of floating point numbers, that is filled by the function with all temperatures (in degrees Celcius) for which the resistance of the thermistor is specified.

resValues array of floating point numbers, that is filled by the function with the value (in Ohms) for each of the temperature included in the first argument, index by index.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→muteValueCallbacks() YTemperature
temperature.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→nextTemperature() YTemperature
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→registerTimedReportCallback() YTemperature
temperature.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

temperature→registerValueCallback() YTemperature
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() YTemperature
temperature→setHighestValue()
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_logFrequency() YTemperature
temperature→setLogFrequency()
temperature.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logicalName() YTemperature
temperature→setLogicalName()
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() YTemperature
temperature→setLowestValue()
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_ntcParameters() YTemperature
temperature→setNtcParameters()
temperature.set_ntcParameters()

Configures NTC thermistor parameters in order to properly compute the temperature from the measured resistance.

function **set_ntcParameters**(**res25**, **beta**)

For increased precision, you can enter a complete mapping table using set_thermistorResponseTable. This function can only be used with a temperature sensor based on thermistors.

Parameters :

res25 thermistor resistance at 25 degrees Celsius

beta Beta value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_reportFrequency() YTemperature
temperature→setReportFrequency()
temperature.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_resolution() YTemperature
temperature→setResolution()
temperature.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_sensorType() YTemperature
temperature→setSensorType()
temperature.set_sensorType()

Modifies the temperature sensor type.

function **set_sensorType**(**newval**)

This function is used to define the type of thermocouple (K,E...) used with the device. It has no effect if module is using a digital sensor or a thermistor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_SENSORTYPE_DIGITAL`, `Y_SENSORTYPE_TYPE_K`, `Y_SENSORTYPE_TYPE_E`, `Y_SENSORTYPE_TYPE_J`, `Y_SENSORTYPE_TYPE_N`, `Y_SENSORTYPE_TYPE_R`, `Y_SENSORTYPE_TYPE_S`, `Y_SENSORTYPE_TYPE_T`, `Y_SENSORTYPE_PT100_4WIRES`, `Y_SENSORTYPE_PT100_3WIRES`, `Y_SENSORTYPE_PT100_2WIRES`, `Y_SENSORTYPE_RES_OHM`, `Y_SENSORTYPE_RES_NTC`, `Y_SENSORTYPE_RES_LINEAR` and `Y_SENSORTYPE_RES_INTERNAL`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_thermistorResponseTable() YTemperature
temperature→setThermistorResponseTable()
temperature.set_thermistorResponseTable()

Records a thermistor response table, in order to interpolate the temperature from the measured resistance.

function **set_thermistorResponseTable**(**tempValues**, **resValues**)

This function can only be used with a temperature sensor based on thermistors.

Parameters :

tempValues array of floating point numbers, corresponding to all temperatures (in degrees Celcius) for which the resistance of the thermistor is specified.

resValues array of floating point numbers, corresponding to the resistance values (in Ohms) for each of the temperature included in the first argument, index by index.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_unit() YTemperature
temperature→setUnit()temperature.set_unit()

Changes the measuring unit for the measured temperature.

function **set_unit**(**newval**)

That unit is a string. If that strings end with the letter F all temperatures values will returned in Fahrenheit degrees. If that String ends with the letter K all values will be returned in Kelvin degrees. If that string ends with the letter C all values will be returned in Celsius degrees. If the string ends with any other character the change will be ignored. Remember to call the `saveToFlash()` method of the module if the modification must be kept. WARNING: if a specific calibration is defined for the temperature function, a unit system change will probably break it.

Parameters :

newval a string corresponding to the measuring unit for the measured temperature

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_userData() YTemperature
temperature→setUserData()
temperature.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

function **set_userData**(**data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

temperature→startDataLogger() YTemperature
temperature.startDataLogger()

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

temperature→stopDataLogger() YTemperature
temperature.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

temperature→unmuteValueCallbacks() YTemperature
temperature.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→wait_async()temperature.wait_async() YTemperature

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.65. Tilt function interface

The YSensor class is the parent class for all Yoctopuce sensors. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provide a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButton class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

```
js    <script type='text/javascript' src='yocto_tilt.js'></script>
cpp   #include "yocto_tilt.h"
m     #import "yocto_tilt.h"
pas   uses yocto_tilt;
vb    yocto_tilt.vb
cs    yocto_tilt.cs
java  import com.yoctopuce.YoctoAPI.YTilt;
uwp   import com.yoctopuce.YoctoAPI.YTilt;
py    from yocto_tilt import *
php   require_once('yocto_tilt.php');
es    in HTML: <script src="../../lib/yocto_tilt.js"></script>
      in node.js: require('yoctolib-es2017/yocto_tilt.js');
```

Global functions
yFindTilt(**func**) Retrieves a tilt sensor for a given identifier.
yFindTiltInContext(**yctx**, **func**) Retrieves a tilt sensor for a given identifier in a YAPI context.
yFirstTilt() Starts the enumeration of tilt sensors currently accessible.
yFirstTiltInContext(**yctx**) Starts the enumeration of tilt sensors currently accessible.
`YTilt` methods
tilt→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
tilt→clearCache() Invalidates the cache.
tilt→describe() Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
tilt→get_advertisedValue() Returns the current value of the tilt sensor (no more than 6 characters).
tilt→get_bandwidth() Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).
tilt→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.
tilt→get_currentValue() Returns the current value of the inclination, in degrees, as a floating point number.

tilt→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

tilt→get_errorMessage()

Returns the error message of the latest error with the tilt sensor.

tilt→get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

tilt→get_friendlyName()

Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

tilt→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

tilt→get_functionId()

Returns the hardware identifier of the tilt sensor, without reference to the module.

tilt→get_hardwareId()

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

tilt→get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

tilt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

tilt→get_logicalName()

Returns the logical name of the tilt sensor.

tilt→get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

tilt→get_module()

Gets the `YModule` object for the device on which the function is located.

tilt→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

tilt→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

tilt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

tilt→get_resolution()

Returns the resolution of the measured values.

tilt→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

tilt→get_unit()

Returns the measuring unit for the inclination.

tilt→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

tilt→isOnline()

Checks if the tilt sensor is currently reachable, without raising any error.

tilt→isOnline_async(**callback**, **context**)

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

tilt→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

tilt→load(msValidity)

Preloads the tilt sensor cache with a specified validity duration.

tilt→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

tilt→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

tilt→load_async(msValidity, callback, context)

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

tilt→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

tilt→nextTilt()

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

tilt→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

tilt→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

tilt→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

tilt→set_highestValue(newval)

Changes the recorded maximal value observed.

tilt→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

tilt→set_logicalName(newval)

Changes the logical name of the tilt sensor.

tilt→set_lowestValue(newval)

Changes the recorded minimal value observed.

tilt→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

tilt→set_resolution(newval)

Changes the resolution of the measured physical values.

tilt→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

tilt→startDataLogger()

Starts the data logger on the device.

tilt→stopDataLogger()

Stops the datalogger on the device.

tilt→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

tilt→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTilt.FindTilt() YTilt
yFindTilt()yFindTilt()

Retrieves a tilt sensor for a given identifier.

```
function FindTilt( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the tilt sensor

Returns :

a `YTilt` object allowing you to drive the tilt sensor.

YTilt.FindTiltInContext() YTilt
yFindTiltInContext()

Retrieves a tilt sensor for a given identifier in a YAPI context.

function **FindTiltInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the tilt sensor

Returns :

a `YTilt` object allowing you to drive the tilt sensor.

YTilt.FirstTilt() YTilt yFirstTilt()yFirstTilt()

Starts the enumeration of tilt sensors currently accessible.

function **FirstTilt**( )

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

Returns :

a pointer to a `YTilt` object, corresponding to the first tilt sensor currently online, or a `null` pointer if there are none.

YTilt.FirstTiltInContext() YTilt
yFirstTiltInContext()

Starts the enumeration of tilt sensors currently accessible.

function **FirstTiltInContext**(**yctx**)

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YTilt` object, corresponding to the first tilt sensor currently online, or a `null` pointer if there are none.

tilt→calibrateFromPoints()tilt.calibrateFromPoints() YTilt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

function **calibrateFromPoints**(**rawValues**, **refValues**)

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→clearCache()tilt.clearCache() YTilt

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the tilt sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

tilt→describe()tilt.describe() YTilt

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the tilt sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

tilt→get_advertisedValue() YTilt
tilt→advertisedValue()tilt.get_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the tilt sensor (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

tilt→get_bandwidth() YTilt
tilt→bandwidth()tilt.get_bandwidth()

Returns the measure update frequency, measured in Hz (Yocto-3D-V2 only).

function **get_bandwidth**( )

Returns :
an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

On failure, throws an exception or returns `Y_BANDWIDTH_INVALID`.

tilt→get_currentRawValue() YTilt
tilt→currentRawValue()tilt.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

function **get_currentRawValue**( )

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

tilt→get_currentValue() YTilt
tilt→currentValue()tilt.get_currentValue()

Returns the current value of the inclination, in degrees, as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the inclination, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

tilt→get_dataLogger() YTilt
tilt→dataLogger()tilt.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

tilt→get_errorMessage() YTilt
tilt→errorMessage()tilt.get_errorMessage()

Returns the error message of the latest error with the tilt sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the tilt sensor object

tilt→get_errorType() YTilt
tilt→errorType()tilt.get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the tilt sensor object

tilt→get_friendlyName() YTilt
tilt→friendlyName()tilt.get_friendlyName()

Returns a global identifier of the tilt sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the tilt sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the tilt sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the tilt sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

tilt→get_functionDescriptor() YTilt
tilt→functionDescriptor()tilt.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

tilt→get_functionId() YTilt
tilt→functionId()tilt.get_functionId()

Returns the hardware identifier of the tilt sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the tilt sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

tilt→get_hardwareId() YTilt
tilt→hardwareId()tilt.get_hardwareId()

Returns the unique hardware identifier of the tilt sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the tilt sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the tilt sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

tilt→get_highestValue() YTilt
tilt→highestValue()tilt.get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

tilt→get_logFrequency() YTilt
tilt→logFrequency()tilt.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

tilt→get_logicalName() YTilt
tilt→logicalName()tilt.get_logicalName()

Returns the logical name of the tilt sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the tilt sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

tilt→get_lowestValue() YTilt
tilt→lowestValue()tilt.get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

tilt→get_module() YTilt
tilt→module()tilt.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

tilt→get_recordedData() YTilt
tilt→recordedData()tilt.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

tilt→get_reportFrequency() YTilt
tilt→reportFrequency()tilt.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

tilt→get_resolution() YTilt
tilt→resolution()tilt.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

tilt→get_sensorState() YTilt
tilt→sensorState()tilt.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

tilt→get_unit() YTilt
tilt→unit()tilt.get_unit()

Returns the measuring unit for the inclination.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns `Y_UNIT_INVALID`.

tilt→get_userData() YTilt
tilt→userData()tilt.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.

tilt→isOnline()tilt.isOnline() YTilt

Checks if the tilt sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

Returns :
`true` if the tilt sensor can be reached, and `false` otherwise

tilt→load()tilt.load() YTilt

Preloads the tilt sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→loadAttribute()tilt.loadAttribute() YTilt

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

tilt→loadCalibrationPoints() YTilt
tilt.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→muteValueCallbacks()tilt.muteValueCallbacks() YTilt

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→nextTilt()tilt.nextTilt() YTilt

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

function **nextTilt**( )

Returns :

a pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a `null` pointer if there are no more tilt sensors to enumerate.

tilt→registerTimedReportCallback() YTilt
tilt.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

tilt→registerValueCallback() YTilt
tilt.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.

tilt→set_bandwidth() YTilt
tilt→setBandwidth()tilt.set_bandwidth()

Changes the measure update frequency, measured in Hz (Yocto-3D-V2 only).

```
function set_bandwidth( newval)
```

When the frequency is lower, the device performs averaging.

Parameters :

newval an integer corresponding to the measure update frequency, measured in Hz (Yocto-3D-V2 only)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_highestValue() YTilt
tilt→setHighestValue()tilt.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.

tilt→set_logFrequency() YTilt
tilt→setLogFrequency()tilt.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logicalName() YTilt
tilt→setLogicalName()tilt.set_logicalName()

Changes the logical name of the tilt sensor.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the tilt sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_lowestValue() YTilt
tilt→setLowestValue()tilt.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.

tilt→set_reportFrequency() YTilt
tilt→setReportFrequency()tilt.set_reportFrequency()

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_resolution() YTilt
tilt→setResolution()tilt.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_userData() YTilt
tilt→setUserData()tilt.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

tilt→startDataLogger()tilt.startDataLogger() YTilt

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

tilt→stopDataLogger()tilt.stopDataLogger() YTilt

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :
YAPI_SUCCESS if the call succeeds.

tilt→unmuteValueCallbacks() YTilt
tilt.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→wait_async()tilt.wait_async() YTilt

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.66. Voc function interface

The Yoctopuce class YVoc allows you to read and configure Yoctopuce Volatile Organic Compound sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_voc.js'></script>
cpp     #include "yocto_voc.h"
m       #import "yocto_voc.h"
pas     uses yocto_voc;
vb      yocto_voc.vb
cs      yocto_voc.cs
java    import com.yoctopuce.YoctoAPI.YVoc;
uwp     import com.yoctopuce.YoctoAPI.YVoc;
py      from yocto_voc import *
php     require_once('yocto_voc.php');
es      in HTML: <script src="../../lib/yocto_voc.js"></script>
        in node.js: require('yoctolib-es2017/yocto_voc.js');
```

Global functions
yFindVoc(**func**) Retrieves a Volatile Organic Compound sensor for a given identifier.
yFindVocInContext(**yctx**, **func**) Retrieves a Volatile Organic Compound sensor for a given identifier in a YAPI context.
yFirstVoc() Starts the enumeration of Volatile Organic Compound sensors currently accessible.
yFirstVocInContext(**yctx**) Starts the enumeration of Volatile Organic Compound sensors currently accessible.
YVoc methods
voc→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
voc→clearCache() Invalidates the cache.
voc→describe() Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
voc→get_advertisedValue() Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).
voc→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.
voc→get_currentValue() Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.
voc→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
voc→get_errorMessage() Returns the error message of the latest error with the Volatile Organic Compound sensor.

voc→get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

voc→get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

voc→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voc→get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

voc→get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

voc→get_highestValue()

Returns the maximal value observed for the estimated VOC concentration since the device was started.

voc→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voc→get_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

voc→get_lowestValue()

Returns the minimal value observed for the estimated VOC concentration since the device was started.

voc→get_module()

Gets the `YModule` object for the device on which the function is located.

voc→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voc→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voc→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voc→get_resolution()

Returns the resolution of the measured values.

voc→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

voc→get_unit()

Returns the measuring unit for the estimated VOC concentration.

voc→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voc→isOnline()

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

voc→isOnline_async(**callback**, **context**)

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

voc→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

voc→load(**msValidity**)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

voc→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

voc→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voc→load_async(msValidity, callback, context)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

voc→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

voc→nextVoc()

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

voc→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voc→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voc→set_highestValue(newval)

Changes the recorded maximal value observed.

voc→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voc→set_logicalName(newval)

Changes the logical name of the Volatile Organic Compound sensor.

voc→set_lowestValue(newval)

Changes the recorded minimal value observed.

voc→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voc→set_resolution(newval)

Changes the resolution of the measured physical values.

voc→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voc→startDataLogger()

Starts the data logger on the device.

voc→stopDataLogger()

Stops the datalogger on the device.

voc→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

voc→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoc.FindVoc() YVoc yFindVoc()yFindVoc()

Retrieves a Volatile Organic Compound sensor for a given identifier.

```
function FindVoc( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Volatile Organic Compound sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoc.isOnline()` to test if the Volatile Organic Compound sensor is indeed online at a given time. In case of ambiguity when looking for a Volatile Organic Compound sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the Volatile Organic Compound sensor

Returns :

a YVoc object allowing you to drive the Volatile Organic Compound sensor.

YVoc.FindVocInContext() YVoc yFindVocInContext()

Retrieves a Volatile Organic Compound sensor for a given identifier in a YAPI context.

function **FindVocInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Volatile Organic Compound sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoc.isOnline()` to test if the Volatile Organic Compound sensor is indeed online at a given time. In case of ambiguity when looking for a Volatile Organic Compound sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the Volatile Organic Compound sensor

Returns :

a YVoc object allowing you to drive the Volatile Organic Compound sensor.

YVoc.FirstVoc() YVoc yFirstVoc()yFirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

function **FirstVoc**( )

Use the method YVoc.nextVoc() to iterate on next Volatile Organic Compound sensors.

Returns :

a pointer to a YVoc object, corresponding to the first Volatile Organic Compound sensor currently online, or a null pointer if there are none.

YVoc.FirstVocInContext() YVoc yFirstVocInContext()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

function **FirstVocInContext**(**yctx**)

Use the method YVoc.nextVoc() to iterate on next Volatile Organic Compound sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a YVoc object, corresponding to the first Volatile Organic Compound sensor currently online, or a null pointer if there are none.

voc→calibrateFromPoints()voc.calibrateFromPoints() YVoc

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

function **calibrateFromPoints**(**rawValues**, **refValues**)

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→clearCache()voc.clearCache() YVoc

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the Volatile Organic Compound sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

voc→describe()voc.describe() YVoc

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the Volatile Organic Compound sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

voc→get_advertisedValue() YVoc
voc→advertisedValue()voc.get_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

voc→get_currentRawValue() YVoc
voc→currentRawValue()voc.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

voc→get_currentValue() YVoc
voc→currentValue()voc.get_currentValue()

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current value of the estimated VOC concentration, in ppm (vol), as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

voc→get_dataLogger() YVoc
voc→dataLogger()voc.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

voc→get_errorMessage() YVoc
voc→errorMessage()voc.get_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the Volatile Organic Compound sensor object

voc→get_errorType() YVoc
voc→errorType()voc.get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

```
function get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the Volatile Organic Compound sensor object

voc→get_friendlyName() YVoc
voc→friendlyName()voc.get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the Volatile Organic Compound sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the Volatile Organic Compound sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

voc→get_functionDescriptor() YVoc
voc→functionDescriptor()
voc.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voc→get_functionId() YVoc
voc→functionId()voc.get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the Volatile Organic Compound sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voc→get_hardwareId() YVoc
voc→hardwareId()voc.get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Volatile Organic Compound sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the Volatile Organic Compound sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

voc→get_highestValue() YVoc
voc→highestValue()voc.get_highestValue()

Returns the maximal value observed for the estimated VOC concentration since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the estimated VOC concentration since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

voc→get_logFrequency() YVoc
voc→logFrequency()voc.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

voc→get_logicalName() YVoc
voc→logicalName()voc.get_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the Volatile Organic Compound sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

voc→get_lowestValue() YVoc
voc→lowestValue()voc.get_lowestValue()

Returns the minimal value observed for the estimated VOC concentration since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the estimated VOC concentration since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voc→get_module() YVoc
voc→module()voc.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

voc→get_recordedData() YVoc
voc→recordedData()voc.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voc→get_reportFrequency() YVoc
voc→reportFrequency()voc.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

voc→get_resolution() YVoc
voc→resolution()voc.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voc→get_sensorState() YVoc
voc→sensorState()voc.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

voc→get_unit() YVoc
voc→unit()voc.get_unit()

Returns the measuring unit for the estimated VOC concentration.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voc→get_userData() YVoc
voc→userData()voc.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.

voc→isOnline()voc.isOnline() YVoc

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Volatile Organic Compound sensor.

Returns :

`true` if the Volatile Organic Compound sensor can be reached, and `false` otherwise

voc→load()voc.load() YVoc

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→loadAttribute()voc.loadAttribute() YVoc

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

voc→loadCalibrationPoints() YVoc
voc.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→muteValueCallbacks() YVoc
voc.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→nextVoc()voc.nextVoc() YVoc

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

function **nextVoc**( )

Returns :

a pointer to a `YVoc` object, corresponding to a Volatile Organic Compound sensor currently online, or a `null` pointer if there are no more Volatile Organic Compound sensors to enumerate.

voc→registerTimedReportCallback() YVoc
voc.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

voc→registerValueCallback() YVoc
voc.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.

voc→set_highestValue() YVoc
voc→setHighestValue()voc.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.

voc→set_logFrequency() YVoc
voc→setLogFrequency()voc.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_logicalName() YVoc
voc→setLogicalName()voc.set_logicalName()

Changes the logical name of the Volatile Organic Compound sensor.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Volatile Organic Compound sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_lowestValue() YVoc
voc→setLowestValue()voc.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.

voc→set_reportFrequency() YVoc
voc→setReportFrequency()
voc.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_resolution() YVoc
voc→setResolution()voc.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_userData() YVoc
voc→setUserData()voc.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

voc→startDataLogger()voc.startDataLogger() YVoc

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

voc→stopDataLogger()voc.stopDataLogger() YVoc

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

voc→unmuteValueCallbacks() YVoc
voc.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→wait_async()voc.wait_async() YVoc

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.67. Voltage function interface

The Yoctopuce class YVoltage allows you to read and configure Yoctopuce voltage sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_voltage.js'></script>
cpp     #include "yocto_voltage.h"
m       #import "yocto_voltage.h"
pas     uses yocto_voltage;
vb      yocto_voltage.vb
cs      yocto_voltage.cs
java    import com.yoctopuce.YoctoAPI.YVoltage;
uwp     import com.yoctopuce.YoctoAPI.YVoltage;
py      from yocto_voltage import *
php     require_once('yocto_voltage.php');
es      in HTML: <script src="../../lib/yocto_voltage.js"></script>
        in node.js: require('yoctolib-es2017/yocto_voltage.js');
```

Global functions
yFindVoltage(**func**) Retrieves a voltage sensor for a given identifier.
yFindVoltageInContext(**yctx**, **func**) Retrieves a voltage sensor for a given identifier in a YAPI context.
yFirstVoltage() Starts the enumeration of voltage sensors currently accessible.
yFirstVoltageInContext(**yctx**) 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→clearCache() Invalidates the cache.
voltage→describe() Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
voltage→get_advertisedValue() Returns the current value of the voltage sensor (no more than 6 characters).
voltage→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.
voltage→get_currentValue() Returns the current value of the voltage, in Volt, as a floating point number.
voltage→get_dataLogger() Returns the YDatalogger object of the device hosting the sensor.
voltage→get_errorMessage() Returns the error message of the latest error with the voltage sensor.
voltage→get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

voltage→get_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

voltage→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voltage→get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

voltage→get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

voltage→get_highestValue()

Returns the maximal value observed for the voltage since the device was started.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage since the device was started.

voltage→get_module()

Gets the `YModule` object for the device on which the function is located.

voltage→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(**callback**, **context**)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

voltage→load(**msValidity**)

Preloads the voltage sensor cache with a specified validity duration.

voltage→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

voltage→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voltage→load_async(msValidity, callback, context)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

voltage→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_highestValue(newval)

Changes the recorded maximal value observed.

voltage→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(newval)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured physical values.

voltage→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltage→startDataLogger()

Starts the data logger on the device.

voltage→stopDataLogger()

Stops the datalogger on the device.

voltage→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltage.FindVoltage() YVoltage yFindVoltage()yFindVoltage()

Retrieves a voltage sensor for a given identifier.

```
function FindVoltage( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FindVoltageInContext() YVoltage yFindVoltageInContext()

Retrieves a voltage sensor for a given identifier in a YAPI context.

function **FindVoltageInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FirstVoltage() YVoltage yFirstVoltage()yFirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

function **FirstVoltage**( )

Use the method YVoltage.nextVoltage() to iterate on next voltage sensors.

Returns :

a pointer to a YVoltage object, corresponding to the first voltage sensor currently online, or a null pointer if there are none.

YVoltage.FirstVoltageInContext() yFirstVoltageInContext()

YVoltage

Starts the enumeration of voltage sensors currently accessible.

function **FirstVoltageInContext**(**yctx**)

Use the method `YVoltage.nextVoltage()` to iterate on next voltage sensors.

Parameters :

yctx a YAPI context.

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() YVoltage
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 linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→clearCache()voltage.clearCache() YVoltage

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the voltage sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

voltage→describe()voltage.describe() YVoltage

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voltage→get_advertisedValue() YVoltage
voltage→advertisedValue()
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() YVoltage
voltage→currentRawValue()
voltage.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

voltage→get_currentValue() YVoltage
voltage→currentValue()voltage.get_currentValue()

Returns the current value of the voltage, in Volt, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the voltage, in Volt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

voltage→get_dataLogger() YVoltage
voltage→dataLogger()voltage.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

voltage→get_errorMessage() YVoltage
voltage→errorMessage()voltage.get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the voltage sensor object

voltage→get_errorType() YVoltage
voltage→errorType()voltage.get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the voltage sensor object

voltage→get_friendlyName() YVoltage
voltage→friendlyName()voltage.get_friendlyName()

Returns a global identifier of the voltage sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the voltage sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the voltage sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

voltage→get_functionDescriptor() YVoltage
voltage→functionDescriptor()
voltage.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voltage→get_functionId() YVoltage
voltage→functionId()voltage.get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the voltage sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voltage→get_hardwareId() YVoltage
voltage→hardwareId()voltage.get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the voltage sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

voltage→get_highestValue() YVoltage
voltage→highestValue()voltage.get_highestValue()

Returns the maximal value observed for the voltage since the device was started.

function **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

voltage→get_logFrequency() YVoltage
voltage→logFrequency()voltage.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

voltage→get_logicalName() YVoltage
voltage→logicalName()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() YVoltage
voltage→lowestValue()voltage.get_lowestValue()

Returns the minimal value observed for the voltage since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voltage→get_module() YVoltage
voltage→module()voltage.get_module()

Gets the YModule object for the device on which the function is located.

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_recordedData() YVoltage
voltage→recordedData()voltage.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voltage→get_reportFrequency() YVoltage
voltage→reportFrequency()
voltage.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

voltage→get_resolution() YVoltage
voltage→resolution()voltage.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voltage→get_sensorState() YVoltage
voltage→sensorState()voltage.get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

voltage→get_unit() YVoltage
voltage→unit()voltage.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`.

voltage→get_userData() YVoltage
voltage→userData()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()voltage.isOnline() YVoltage

Checks if the voltage sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

Returns :

`true` if the voltage sensor can be reached, and `false` otherwise

voltage→load()voltage.load() YVoltage

Preloads the voltage sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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→loadAttribute()voltage.loadAttribute() YVoltage

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

voltage→loadCalibrationPoints() YVoltage
voltage.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→muteValueCallbacks() YVoltage
voltage.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→nextVoltage()voltage.nextVoltage() YVoltage

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

function **nextVoltage**( )

Returns :

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a `null` pointer if there are no more voltage sensors to enumerate.

voltage→registerTimedReportCallback() YVoltage
voltage.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

voltage→registerValueCallback() YVoltage
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() YVoltage
voltage→setHighestValue()
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_logFrequency() YVoltage
voltage→setLogFrequency()
voltage.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_logicalName() YVoltage
voltage→setLogicalName()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() YVoltage
voltage→setLowestValue()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_reportFrequency() YVoltage
voltage→setReportFrequency()
voltage.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_resolution() YVoltage
voltage→setResolution()voltage.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_userData() YVoltage
voltage→setUserData()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

voltage→startDataLogger()voltage.startDataLogger() YVoltage

Starts the data logger on the device.

function **startDataLogger**( )

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

voltage→stopDataLogger()voltage.stopDataLogger() YVoltage

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

voltage→unmuteValueCallbacks() YVoltage
voltage.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→wait_async()voltage.wait_async() YVoltage

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.68. VoltageOutput function interface

The Yoctopuce application programming interface allows you to change the value of the voltage output.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voltageoutput.js'></script>
cpp	#include "yocto_voltageoutput.h"
m	#import "yocto_voltageoutput.h"
pas	uses yocto_voltageoutput;
vb	yocto_voltageoutput.vb
cs	yocto_voltageoutput.cs
java	import com.yoctopuce.YoctoAPI.YVoltageOutput;
uwp	import com.yoctopuce.YoctoAPI.YVoltageOutput;
py	from yocto_voltageoutput import *
php	require_once('yocto_voltageoutput.php');
es	in HTML: <script src="../../lib/yocto_voltageoutput.js"></script> in node.js: require('yoctolib-es2017/yocto_voltageoutput.js');

Global functions

yFindVoltageOutput(**func**)

Retrieves a voltage output for a given identifier.

yFindVoltageOutputInContext(**yctx**, **func**)

Retrieves a voltage output for a given identifier in a YAPI context.

yFirstVoltageOutput()

Starts the enumeration of voltage outputs currently accessible.

yFirstVoltageOutputInContext(**yctx**)

Starts the enumeration of voltage outputs currently accessible.

YVoltageOutput methods

voltageoutput→clearCache()

Invalidates the cache.

voltageoutput→describe()

Returns a short text that describes unambiguously the instance of the voltage output in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

voltageoutput→get_advertisedValue()

Returns the current value of the voltage output (no more than 6 characters).

voltageoutput→get_currentVoltage()

Returns the output voltage set point, in V

voltageoutput→get_errorMessage()

Returns the error message of the latest error with the voltage output.

voltageoutput→get_errorType()

Returns the numerical error code of the latest error with the voltage output.

voltageoutput→get_friendlyName()

Returns a global identifier of the voltage output in the format `MODULE_NAME.FUNCTION_NAME`.

voltageoutput→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voltageoutput→get_functionId()

Returns the hardware identifier of the voltage output, without reference to the module.

voltageoutput→get_hardwareId()

Returns the unique hardware identifier of the voltage output in the form `SERIAL.FUNCTIONID`.

voltageoutput→get_logicalName()

Returns the logical name of the voltage output.

voltageoutput→get_module()

Gets the `YModule` object for the device on which the function is located.

voltageoutput→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voltageoutput→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voltageoutput→get_voltageAtStartUp()

Returns the selected voltage output at device startup, in V.

voltageoutput→isOnline()

Checks if the voltage output is currently reachable, without raising any error.

voltageoutput→isOnline_async(callback, context)

Checks if the voltage output is currently reachable, without raising any error (asynchronous version).

voltageoutput→load(msValidity)

Preloads the voltage output cache with a specified validity duration.

voltageoutput→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

voltageoutput→load_async(msValidity, callback, context)

Preloads the voltage output cache with a specified validity duration (asynchronous version).

voltageoutput→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

voltageoutput→nextVoltageOutput()

Continues the enumeration of voltage outputs started using `yFirstVoltageOutput()`.

voltageoutput→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voltageoutput→set_currentVoltage(newval)

Changes the output voltage, in V.

voltageoutput→set_logicalName(newval)

Changes the logical name of the voltage output.

voltageoutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltageoutput→set_voltageAtStartUp(newval)

Changes the output voltage at device start up.

voltageoutput→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

voltageoutput→voltageMove(V_target, ms_duration)

Performs a smooth transistion of output voltage.

voltageoutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltageOutput.FindVoltageOutput() YVoltageOutput yFindVoltageOutput()yFindVoltageOutput()

Retrieves a voltage output for a given identifier.

function **FindVoltageOutput**(**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 output is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltageOutput.isOnline()` to test if the voltage output is indeed online at a given time. In case of ambiguity when looking for a voltage output 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the voltage output

Returns :

a `YVoltageOutput` object allowing you to drive the voltage output.

YVoltageOutput.FindVoltageOutputInContext() YVoltageOutput yFindVoltageOutputInContext()

Retrieves a voltage output for a given identifier in a YAPI context.

function **FindVoltageOutputInContext**(**yctx**, **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 output is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltageOutput.isOnline()` to test if the voltage output is indeed online at a given time. In case of ambiguity when looking for a voltage output 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 :

yctx a YAPI context

func a string that uniquely characterizes the voltage output

Returns :

a `YVoltageOutput` object allowing you to drive the voltage output.

YVoltageOutput.FirstVoltageOutput() YVoltageOutput yFirstVoltageOutput()yFirstVoltageOutput()

Starts the enumeration of voltage outputs currently accessible.

function **FirstVoltageOutput**( )

Use the method `YVoltageOutput.nextVoltageOutput()` to iterate on next voltage outputs.

Returns :

a pointer to a `YVoltageOutput` object, corresponding to the first voltage output currently online, or a `null` pointer if there are none.

YVoltageOutput.FirstVoltageOutputInContext() YVoltageOutput yFirstVoltageOutputInContext()

Starts the enumeration of voltage outputs currently accessible.

function **FirstVoltageOutputInContext**(**yctx**)

Use the method `YVoltageOutput.nextVoltageOutput()` to iterate on next voltage outputs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YVoltageOutput` object, corresponding to the first voltage output currently online, or a `null` pointer if there are none.

voltageoutput→clearCache() YVoltageOutput
voltageoutput.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the voltage output attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

voltageoutput→describe()voltageoutput.describe() YVoltageOutput

Returns a short text that describes unambiguously the instance of the voltage output in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage output (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voltageoutput→get_advertisedValue() YVoltageOutput
voltageoutput→advertisedValue()
voltageoutput.get_advertisedValue()

Returns the current value of the voltage output (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the voltage output (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

voltageoutput→get_currentVoltage() YVoltageOutput
voltageoutput→currentVoltage()
voltageoutput.get_currentVoltage()

Returns the output voltage set point, in V

function **get_currentVoltage**( )

Returns :
a floating point number corresponding to the output voltage set point, in V

On failure, throws an exception or returns `Y_CURRENTVOLTAGE_INVALID`.

voltageoutput→get_errorMessage() YVoltageOutput
voltageoutput→errorMessage()
voltageoutput.get_errorMessage()

Returns the error message of the latest error with the voltage output.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the voltage output object

voltageoutput→get_errorType() YVoltageOutput
voltageoutput→errorType()
voltageoutput.get_errorType()

Returns the numerical error code of the latest error with the voltage output.

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 the voltage output object

voltageoutput→get_friendlyName() YVoltageOutput
voltageoutput→friendlyName()
voltageoutput.get_friendlyName()

Returns a global identifier of the voltage output in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the voltage output if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage output (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage output using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

voltageoutput→get_functionDescriptor() YVoltageOutput
voltageoutput→functionDescriptor()
voltageoutput.get_functionDescriptor()

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.

voltageoutput→get_functionId() YVoltageOutput
voltageoutput→functionId()
voltageoutput.get_functionId()

Returns the hardware identifier of the voltage output, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the voltage output (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voltageoutput→get_hardwareId() YVoltageOutput
voltageoutput→hardwareId()
voltageoutput.get_hardwareId()

Returns the unique hardware identifier of the voltage output in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage output (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the voltage output (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

voltageoutput→get_logicalName() YVoltageOutput
voltageoutput→logicalName()
voltageoutput.get_logicalName()

Returns the logical name of the voltage output.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the voltage output.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

voltageoutput→get_module() YVoltageOutput
voltageoutput→module()voltageoutput.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

voltageoutput→get_userData() YVoltageOutput
voltageoutput→userData()
voltageoutput.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.

voltageoutput→get_voltageAtStartUp() YVoltageOutput
voltageoutput→voltageAtStartUp()
voltageoutput.get_voltageAtStartUp()

Returns the selected voltage output at device startup, in V.

function **get_voltageAtStartUp**( )

Returns :
a floating point number corresponding to the selected voltage output at device startup, in V

On failure, throws an exception or returns `Y_VOLTAGEATSTARTUP_INVALID`.

voltageoutput→isOnline()voltageoutput.isOnline() YVoltageOutput

Checks if the voltage output is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the voltage output in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage output.

Returns :
`true` if the voltage output can be reached, and `false` otherwise

voltageoutput→load()voltageoutput.load() YVoltageOutput

Preloads the voltage output 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.

voltageoutput→loadAttribute() YVoltageOutput
voltageoutput.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

voltageoutput→muteValueCallbacks() YVoltageOutput
voltageoutput.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltageoutput→nextVoltageOutput() YVoltageOutput
voltageoutput.nextVoltageOutput()

Continues the enumeration of voltage outputs started using `yFirstVoltageOutput()`.

function **nextVoltageOutput**( )

Returns :

a pointer to a `YVoltageOutput` object, corresponding to a voltage output currently online, or a `null` pointer if there are no more voltage outputs to enumerate.

voltageoutput→registerValueCallback() YVoltageOutput
voltageoutput.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.

voltageoutput→set_currentVoltage() YVoltageOutput
voltageoutput→setCurrentVoltage()
voltageoutput.set_currentVoltage()

Changes the output voltage, in V.

function **set_currentVoltage**(**newval**)

Valid range is from 0 to 10V.

Parameters :

newval a floating point number corresponding to the output voltage, in V

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltageoutput→set_logicalName() YVoltageOutput
voltageoutput→setLogicalName()
voltageoutput.set_logicalName()

Changes the logical name of the voltage output.

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 output.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltageoutput→set_userData() YVoltageOutput
voltageoutput→setUserData()
voltageoutput.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

voltageoutput→set_voltageAtStartUp() YVoltageOutput
voltageoutput→setVoltageAtStartUp()
voltageoutput.set_voltageAtStartUp()

Changes the output voltage at device start up.

function **set_voltageAtStartUp**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call has no effect.

Parameters :

newval a floating point number corresponding to the output voltage at device start up

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltageoutput→unmuteValueCallbacks() YVoltageOutput
voltageoutput.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltageoutput→voltageMove() YVoltageOutput
voltageoutput.voltageMove()

Performs a smooth transistion of output voltage.

function **voltageMove**(**V_target**, **ms_duration**)

Any explicit voltage change cancels any ongoing transition process.

Parameters :

V_target	new output voltage value at the end of the transition (floating-point number, representing the end voltage in V)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

voltageoutput→wait_async() YVoltageOutput
voltageoutput.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.69. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

```
js    <script type='text/javascript' src='yocto_vsource.js'></script>
php   require_once('yocto_vsource.php');
cpp   #include "yocto_vsource.h"
m     #import "yocto_vsource.h"
pas   uses yocto_vsource;
vb    yocto_vsource.vb
cs    yocto_vsource.cs
java  import com.yoctopuce.YoctoAPI.YVSource;
py    from yocto_vsource import *
```

Global functions
yFindVSource(**func**) Retrieves a voltage source for a given identifier.
yFirstVSource() Starts the enumeration of voltage sources currently accessible.
YVSource methods
vsource→describe() Returns a short text that describes the function in the form TYPE(NAME)=SERIAL.FUNCTIONID.
vsource→get_advertisedValue() Returns the current value of the voltage source (no more than 6 characters).
vsource→get_errorMessage() Returns the error message of the latest error with this function.
vsource→get_errorType() Returns the numerical error code of the latest error with this function.
vsource→get_extPowerFailure() Returns true if external power supply voltage is too low.
vsource→get_failure() Returns true if the module is in failure mode.
vsource→get_friendlyName() Returns a global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.
vsource→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
vsource→get_functionId() Returns the hardware identifier of the function, without reference to the module.
vsource→get_hardwareId() Returns the unique hardware identifier of the function in the form SERIAL.FUNCTIONID.
vsource→get_logicalName() Returns the logical name of the voltage source.
vsource→get_module() Gets the YModule object for the device on which the function is located.
vsource→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

vsource→get_overCurrent()

Returns true if the appliance connected to the device is too greedy .

vsource→get_overHeat()

Returns TRUE if the module is overheating.

vsource→get_overLoad()

Returns true if the device is not able to maintaint the requested voltage output .

vsource→get_regulationFailure()

Returns true if the voltage output is too high regarding the requested voltage .

vsource→get_unit()

Returns the measuring unit for the voltage.

vsource→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

vsource→get_voltage()

Returns the voltage output command (mV)

vsource→isOnline()

Checks if the function is currently reachable, without raising any error.

vsource→isOnline_async(**callback**, **context**)

Checks if the function is currently reachable, without raising any error (asynchronous version).

vsource→load(**msValidity**)

Preloads the function cache with a specified validity duration.

vsource→load_async(**msValidity**, **callback**, **context**)

Preloads the function cache with a specified validity duration (asynchronous version).

vsource→nextVSource()

Continues the enumeration of voltage sources started using yFirstVSource().

vsource→pulse(**voltage**, **ms_duration**)

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

vsource→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

vsource→set_logicalName(**newval**)

Changes the logical name of the voltage source.

vsource→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

vsource→set_voltage(**newval**)

Tunes the device output voltage (milliVolts).

vsource→voltageMove(**target**, **ms_duration**)

Performs a smooth move at constant speed toward a given value.

vsource→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

3.70. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
cpp	#include "yocto_wakeupmonitor.h"
m	#import "yocto_wakeupmonitor.h"
pas	uses yocto_wakeupmonitor;
vb	yocto_wakeupmonitor.vb
cs	yocto_wakeupmonitor.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
uwp	import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py	from yocto_wakeupmonitor import *
php	require_once('yocto_wakeupmonitor.php');
es	in HTML: <script src="../../lib/yocto_wakeupmonitor.js"></script> in node.js: require('yoctolib-es2017/yocto_wakeupmonitor.js');

Global functions

yFindWakeUpMonitor(**func**)

Retrieves a monitor for a given identifier.

yFindWakeUpMonitorInContext(**yctx**, **func**)

Retrieves a monitor for a given identifier in a YAPI context.

yFirstWakeUpMonitor()

Starts the enumeration of monitors currently accessible.

yFirstWakeUpMonitorInContext(**yctx**)

Starts the enumeration of monitors currently accessible.

`YWakeUpMonitor` methods

wakeupmonitor→clearCache()

Invalidates the cache.

wakeupmonitor→describe()

Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wakeupmonitor→get_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

wakeupmonitor→get_errorMessage()

Returns the error message of the latest error with the monitor.

wakeupmonitor→get_errorType()

Returns the numerical error code of the latest error with the monitor.

wakeupmonitor→get_friendlyName()

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

wakeupmonitor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

wakeupmonitor→get_functionId()

Returns the hardware identifier of the monitor, without reference to the module.

wakeupmonitor→get_hardwareId()

Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.

wakeupmonitor→get_logicalName()

Returns the logical name of the monitor.

wakeupmonitor→get_module()

Gets the `YModule` object for the device on which the function is located.

wakeupmonitor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

wakeupmonitor→get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format).

wakeupmonitor→get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

wakeupmonitor→get_sleepCountdown()

Returns the delay before the next sleep period.

wakeupmonitor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupmonitor→get_wakeUpReason()

Returns the latest wake up reason.

wakeupmonitor→get_wakeUpState()

Returns the current state of the monitor.

wakeupmonitor→isOnline()

Checks if the monitor is currently reachable, without raising any error.

wakeupmonitor→isOnline_async(**callback**, **context**)

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

wakeupmonitor→load(**msValidity**)

Preloads the monitor cache with a specified validity duration.

wakeupmonitor→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

wakeupmonitor→load_async(**msValidity**, **callback**, **context**)

Preloads the monitor cache with a specified validity duration (asynchronous version).

wakeupmonitor→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

wakeupmonitor→nextWakeUpMonitor()

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

wakeupmonitor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

wakeupmonitor→resetSleepCountDown()

Resets the sleep countdown.

wakeupmonitor→set_logicalName(**newval**)

Changes the logical name of the monitor.

wakeupmonitor→set_nextWakeUp(**newval**)

Changes the days of the week when a wake up must take place.

wakeupmonitor→set_powerDuration(**newval**)

Changes the maximal wake up time (seconds) before automatically going to sleep.

wakeupmonitor→set_sleepCountdown(**newval**)

Changes the delay before the next sleep period.

wakeupmonitor→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

wakeupmonitor→sleep(secBeforeSleep)

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep)

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep)

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

wakeupmonitor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wakeupmonitor→wakeUp()

Forces a wake up.

YWakeUpMonitor.FindWakeUpMonitor() YWakeUpMonitor yFindWakeUpMonitor()yFindWakeUpMonitor()

Retrieves a monitor for a given identifier.

function **FindWakeUpMonitor**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpMonitor.isOnline()` to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the monitor

Returns :

a `YWakeUpMonitor` object allowing you to drive the monitor.

YWakeUpMonitor.FindWakeUpMonitorInContext() YWakeUpMonitor yFindWakeUpMonitorInContext()

Retrieves a monitor for a given identifier in a YAPI context.

function **FindWakeUpMonitorInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpMonitor.isOnline()` to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the monitor

Returns :

a `YWakeUpMonitor` object allowing you to drive the monitor.

YWakeUpMonitor.FirstWakeUpMonitor() YWakeUpMonitor yFirstWakeUpMonitor()yFirstWakeUpMonitor()

Starts the enumeration of monitors currently accessible.

function **FirstWakeUpMonitor**( )

Use the method `YWakeUpMonitor.nextWakeUpMonitor()` to iterate on next monitors.

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to the first monitor currently online, or a `null` pointer if there are none.

YWakeUpMonitor.FirstWakeUpMonitorInContext() YWakeUpMonitor yFirstWakeUpMonitorInContext()

Starts the enumeration of monitors currently accessible.

function **FirstWakeUpMonitorInContext**(**yctx**)

Use the method `YWakeUpMonitor.nextWakeUpMonitor()` to iterate on next monitors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to the first monitor currently online, or a `null` pointer if there are none.

wakeupmonitor→clearCache() YWakeUpMonitor
wakeupmonitor.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the monitor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

wakeupmonitor→describe() YWakeUpMonitor
wakeupmonitor.describe()

Returns a short text that describes unambiguously the instance of the monitor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the monitor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

wakeupmonitor→get_advertisedValue() YWakeUpMonitor
wakeupmonitor→advertisedValue()
wakeupmonitor.get_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the monitor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wakeupmonitor→get_errorMessage() YWakeUpMonitor
wakeupmonitor→errorMessage()
wakeupmonitor.get_errorMessage()

Returns the error message of the latest error with the monitor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the monitor object

wakeupmonitor→get_errorType() YWakeUpMonitor
wakeupmonitor→errorType()
wakeupmonitor.get_errorType()

Returns the numerical error code of the latest error with the monitor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the monitor object

wakeupmonitor→get_friendlyName() YWakeUpMonitor
wakeupmonitor→friendlyName()
wakeupmonitor.get_friendlyName()

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the monitor if they are defined, otherwise the serial number of the module and the hardware identifier of the monitor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the monitor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wakeupmonitor→get_functionDescriptor() YWakeUpMonitor
wakeupmonitor→functionDescriptor()
wakeupmonitor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupmonitor→get_functionId() YWakeUpMonitor
wakeupmonitor→functionId()
wakeupmonitor.get_functionId()

Returns the hardware identifier of the monitor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the monitor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupmonitor→get_hardwareId() YWakeUpMonitor
wakeupmonitor→hardwareId()
wakeupmonitor.get_hardwareId()

Returns the unique hardware identifier of the monitor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the monitor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the monitor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wakeupmonitor→get_logicalName() YWakeUpMonitor
wakeupmonitor→logicalName()
wakeupmonitor.get_logicalName()

Returns the logical name of the monitor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the monitor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupmonitor→get_module() YWakeUpMonitor
wakeupmonitor→module()
wakeupmonitor.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

wakeupmonitor→get_nextWakeUp() YWakeUpMonitor
wakeupmonitor→nextWakeUp()
wakeupmonitor.get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format).

function **get_nextWakeUp**( )

Returns :
an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns `Y_NEXTWAKEUP_INVALID`.

wakeupmonitor→get_powerDuration() YWakeUpMonitor
wakeupmonitor→powerDuration()
wakeupmonitor.get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

function **get_powerDuration**( )

Returns :
an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns `Y_POWERDURATION_INVALID`.

wakeupmonitor→get_sleepCountdown() YWakeUpMonitor
wakeupmonitor→sleepCountdown()
wakeupmonitor.get_sleepCountdown()

Returns the delay before the next sleep period.

function **get_sleepCountdown**( )

Returns :
an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns `Y_SLEEPCOUNTDOWN_INVALID`.

wakeupmonitor→get_userData() YWakeUpMonitor
wakeupmonitor→userData()
wakeupmonitor.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.

wakeupmonitor→get_wakeUpReason() YWakeUpMonitor
wakeupmonitor→wakeUpReason()
wakeupmonitor.get_wakeUpReason()

Returns the latest wake up reason.

function **get_wakeUpReason**( )

Returns :

a value among Y_WAKEUPREASON_USBPOWER, Y_WAKEUPREASON_EXTPOWER, Y_WAKEUPREASON_ENDOFSLEEP, Y_WAKEUPREASON_EXTSIG1, Y_WAKEUPREASON_SCHEDULE1 and Y_WAKEUPREASON_SCHEDULE2 corresponding to the latest wake up reason

On failure, throws an exception or returns Y_WAKEUPREASON_INVALID.

wakeupmonitor→get_wakeUpState() YWakeUpMonitor
wakeupmonitor→wakeUpState()
wakeupmonitor.get_wakeUpState()

Returns the current state of the monitor.

function **get_wakeUpState**( )

Returns :

either Y_WAKEUPSTATE_SLEEPING or Y_WAKEUPSTATE_AWAKE, according to the current state of the monitor

On failure, throws an exception or returns Y_WAKEUPSTATE_INVALID.

wakeupmonitor→isOnline()wakeupmonitor.isOnline() YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

Returns :

`true` if the monitor can be reached, and `false` otherwise

wakeupmonitor→load()wakeupmonitor.load() YWakeUpMonitor

Preloads the monitor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→loadAttribute() YWakeUpMonitor
wakeupmonitor.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

wakeupmonitor→muteValueCallbacks() YWakeUpMonitor
wakeupmonitor.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→nextWakeUpMonitor() YWakeUpMonitor
wakeupmonitor.nextWakeUpMonitor()

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

function **nextWakeUpMonitor**( )

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a `null` pointer if there are no more monitors to enumerate.

wakeupmonitor→registerValueCallback() YWakeUpMonitor
wakeupmonitor.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.

wakeupmonitor→resetSleepCountDown() YWakeUpMonitor
wakeupmonitor.resetSleepCountDown()

Resets the sleep countdown.

function **resetSleepCountDown**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_logicalName() YWakeUpMonitor
wakeupmonitor→setLogicalName()
wakeupmonitor.set_logicalName()

Changes the logical name of the monitor.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the monitor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_nextWakeUp() YWakeUpMonitor
wakeupmonitor→setNextWakeUp()
wakeupmonitor.set_nextWakeUp()

Changes the days of the week when a wake up must take place.

function **set_nextWakeUp**(**newval**)

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_powerDuration() YWakeUpMonitor
wakeupmonitor→setPowerDuration()
wakeupmonitor.set_powerDuration()

Changes the maximal wake up time (seconds) before automatically going to sleep.

function **set_powerDuration**(**newval**)

Parameters :

newval an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_sleepCountdown() YWakeUpMonitor
wakeupmonitor→setSleepCountdown()
wakeupmonitor.set_sleepCountdown()

Changes the delay before the next sleep period.

function **set_sleepCountdown**(**newval**)

Parameters :

newval an integer corresponding to the delay before the next sleep period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_userData() YWakeUpMonitor
wakeupmonitor→setUserData()
wakeupmonitor.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

wakeupmonitor→sleep()wakeupmonitor.sleep() YWakeUpMonitor

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

function **sleep**(**secBeforeSleep**)

Parameters :

secBeforeSleep number of seconds before going into sleep mode,

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepFor() YWakeUpMonitor
wakeupmonitor.sleepFor()

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

function **sleepFor**(**secUntilWakeUp**, **secBeforeSleep**)

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

secUntilWakeUp number of seconds before next wake up

secBeforeSleep number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepUntil() YWakeUpMonitor
wakeupmonitor.sleepUntil()

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

function **sleepUntil**(**wakeUpTime**, **secBeforeSleep**)

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

wakeUpTime	wake-up datetime (UNIX format)
secBeforeSleep	number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→unmuteValueCallbacks() YWakeUpMonitor
wakeupmonitor.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→wait_async() YWakeUpMonitor
wakeupmonitor.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

wakeupmonitor→wakeUp()wakeupmonitor.wakeUp() YWakeUpMonitor

Forces a wake up.

function **wakeUp**( )

3.71. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupschedule.js'></script>
cpp	#include "yocto_wakeupschedule.h"
m	#import "yocto_wakeupschedule.h"
pas	uses yocto_wakeupschedule;
vb	yocto_wakeupschedule.vb
cs	yocto_wakeupschedule.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
uwp	import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
py	from yocto_wakeupschedule import *
php	require_once('yocto_wakeupschedule.php');
es	in HTML: <script src="../../lib/yocto_wakeupschedule.js"></script> in node.js: require('yoctolib-es2017/yocto_wakeupschedule.js');

Global functions

yFindWakeUpSchedule(**func**)

Retrieves a wake up schedule for a given identifier.

yFindWakeUpScheduleInContext(**yctx**, **func**)

Retrieves a wake up schedule for a given identifier in a YAPI context.

yFirstWakeUpSchedule()

Starts the enumeration of wake up schedules currently accessible.

yFirstWakeUpScheduleInContext(**yctx**)

Starts the enumeration of wake up schedules currently accessible.

YWakeUpSchedule methods

wakeupschedule→clearCache()

Invalidates the cache.

wakeupschedule→describe()

Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wakeupschedule→get_advertisedValue()

Returns the current value of the wake up schedule (no more than 6 characters).

wakeupschedule→get_errorMessage()

Returns the error message of the latest error with the wake up schedule.

wakeupschedule→get_errorType()

Returns the numerical error code of the latest error with the wake up schedule.

wakeupschedule→get_friendlyName()

Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

wakeupschedule→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

wakeupschedule→get_functionId()

Returns the hardware identifier of the wake up schedule, without reference to the module.

wakeupschedule→get_hardwareId()

Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.

wakeupschedule→get_hours()
Returns the hours scheduled for wake up.

wakeupschedule→get_logicalName()
Returns the logical name of the wake up schedule.

wakeupschedule→get_minutes()
Returns all the minutes of each hour that are scheduled for wake up.

wakeupschedule→get_minutesA()
Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

wakeupschedule→get_minutesB()
Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

wakeupschedule→get_module()
Gets the `YModule` object for the device on which the function is located.

wakeupschedule→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

wakeupschedule→get_monthDays()
Returns the days of the month scheduled for wake up.

wakeupschedule→get_months()
Returns the months scheduled for wake up.

wakeupschedule→get_nextOccurence()
Returns the date/time (seconds) of the next wake up occurence.

wakeupschedule→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupschedule→get_weekDays()
Returns the days of the week scheduled for wake up.

wakeupschedule→isOnline()
Checks if the wake up schedule is currently reachable, without raising any error.

wakeupschedule→isOnline_async(**callback**, **context**)
Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

wakeupschedule→load(**msValidity**)
Preloads the wake up schedule cache with a specified validity duration.

wakeupschedule→loadAttribute(**attrName**)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

wakeupschedule→load_async(**msValidity**, **callback**, **context**)
Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

wakeupschedule→muteValueCallbacks()
Disables the propagation of every new advertised value to the parent hub.

wakeupschedule→nextWakeUpSchedule()
Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

wakeupschedule→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

wakeupschedule→set_hours(**newval**)
Changes the hours when a wake up must take place.

wakeupschedule→set_logicalName(**newval**)
Changes the logical name of the wake up schedule.

wakeupschedule→set_minutes(**bitmap**)

Changes all the minutes where a wake up must take place.

wakeupschedule→set_minutesA(**newval**)

Changes the minutes in the 00-29 interval when a wake up must take place.

wakeupschedule→set_minutesB(**newval**)

Changes the minutes in the 30-59 interval when a wake up must take place.

wakeupschedule→set_monthDays(**newval**)

Changes the days of the month when a wake up must take place.

wakeupschedule→set_months(**newval**)

Changes the months when a wake up must take place.

wakeupschedule→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

wakeupschedule→set_weekDays(**newval**)

Changes the days of the week when a wake up must take place.

wakeupschedule→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

wakeupschedule→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWakeUpSchedule.FindWakeUpSchedule() YWakeUpSchedule yFindWakeUpSchedule()yFindWakeUpSchedule()

Retrieves a wake up schedule for a given identifier.

function **FindWakeUpSchedule**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpSchedule.isOnline()` to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the wake up schedule

Returns :

a `YWakeUpSchedule` object allowing you to drive the wake up schedule.

YWakeUpSchedule.FindWakeUpScheduleInContext() YWakeUpSchedule yFindWakeUpScheduleInContext()

Retrieves a wake up schedule for a given identifier in a YAPI context.

function **FindWakeUpScheduleInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpSchedule.isOnline()` to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the wake up schedule

Returns :

a `YWakeUpSchedule` object allowing you to drive the wake up schedule.

YWakeUpSchedule.FirstWakeUpSchedule() YWakeUpSchedule yFirstWakeUpSchedule()yFirstWakeUpSchedule()

Starts the enumeration of wake up schedules currently accessible.

function **FirstWakeUpSchedule**( )

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to the first wake up schedule currently online, or a `null` pointer if there are none.

YWakeUpSchedule.FirstWakeUpScheduleInContext() YWakeUpSchedule yFirstWakeUpScheduleInContext()

Starts the enumeration of wake up schedules currently accessible.

function **FirstWakeUpScheduleInContext**(**yctx**)

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to the first wake up schedule currently online, or a `null` pointer if there are none.

wakeupschedule→clearCache() YWakeUpSchedule
wakeupschedule.clearCache()

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the wake up schedule attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

wakeupschedule→describe() YWakeUpSchedule
wakeupschedule.describe()

Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the wake up schedule (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupschedule→get_advertisedValue() YWakeUpSchedule
wakeupschedule→advertisedValue()
wakeupschedule.get_advertisedValue()

Returns the current value of the wake up schedule (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the wake up schedule (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wakeupschedule→get_errorMessage() YWakeUpSchedule
wakeupschedule→errorMessage()
wakeupschedule.get_errorMessage()

Returns the error message of the latest error with the wake up schedule.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the wake up schedule object

wakeupschedule→get_errorType() YWakeUpSchedule
wakeupschedule→errorType()
wakeupschedule.get_errorType()

Returns the numerical error code of the latest error with the wake up schedule.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the wake up schedule object

wakeupschedule→get_friendlyName() YWakeUpSchedule
wakeupschedule→friendlyName()
wakeupschedule.get_friendlyName()

Returns a global identifier of the wake up schedule in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the wake up schedule if they are defined, otherwise the serial number of the module and the hardware identifier of the wake up schedule (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the wake up schedule using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wakeupschedule→get_functionDescriptor() YWakeUpSchedule
wakeupschedule→functionDescriptor()
wakeupschedule.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupschedule→get_functionId() YWakeUpSchedule
wakeupschedule→functionId()
wakeupschedule.get_functionId()

Returns the hardware identifier of the wake up schedule, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the wake up schedule (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupschedule→get_hardwareId() YWakeUpSchedule
wakeupschedule→hardwareId()
wakeupschedule.get_hardwareId()

Returns the unique hardware identifier of the wake up schedule in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wake up schedule (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the wake up schedule (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wakeupschedule→get_hours() YWakeUpSchedule
wakeupschedule→hours()
wakeupschedule.get_hours()

Returns the hours scheduled for wake up.

function **get_hours**( )

Returns :
an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns `Y_HOURS_INVALID`.

wakeupschedule→get_logicalName() YWakeUpSchedule
wakeupschedule→logicalName()
wakeupschedule.get_logicalName()

Returns the logical name of the wake up schedule.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the wake up schedule.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupschedule→get_minutes() YWakeUpSchedule
wakeupschedule→minutes()
wakeupschedule.get_minutes()

Returns all the minutes of each hour that are scheduled for wake up.

```
function get_minutes( )
```

wakeupschedule→get_minutesA() YWakeUpSchedule
wakeupschedule→minutesA()
wakeupschedule.get_minutesA()

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

function **get_minutesA**( )

Returns :
an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESA_INVALID`.

wakeupschedule→get_minutesB() YWakeUpSchedule
wakeupschedule→minutesB()
wakeupschedule.get_minutesB()

Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

function **get_minutesB**( )

Returns :
an integer corresponding to the minutes in the 30-59 intervalof each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESB_INVALID`.

wakeupschedule→get_module() YWakeUpSchedule
wakeupschedule→module()
wakeupschedule.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

wakeupschedule→get_monthDays() YWakeUpSchedule
wakeupschedule→monthDays()
wakeupschedule.get_monthDays()

Returns the days of the month scheduled for wake up.

function **get_monthDays**( )

Returns :
an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns `Y_MONTHDAYS_INVALID`.

wakeupschedule→get_months() YWakeUpSchedule
wakeupschedule→months()
wakeupschedule.get_months()

Returns the months scheduled for wake up.

function **get_months**( )

Returns :
an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns `Y_MONTHS_INVALID`.

wakeupschedule→get_nextOccurence() YWakeUpSchedule
wakeupschedule→nextOccurence()
wakeupschedule.get_nextOccurence()

Returns the date/time (seconds) of the next wake up occurence.

function **get_nextOccurence**( )

Returns :
an integer corresponding to the date/time (seconds) of the next wake up occurence

On failure, throws an exception or returns `Y_NEXTOCCURENCE_INVALID`.

wakeupschedule→get_userData() YWakeUpSchedule
wakeupschedule→userData()
wakeupschedule.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.

wakeupschedule→get_weekDays() YWakeUpSchedule
wakeupschedule→weekDays()
wakeupschedule.get_weekDays()

Returns the days of the week scheduled for wake up.

function **get_weekDays**( )

Returns :
an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns `Y_WEEKDAYS_INVALID`.

wakeupschedule→isOnline() YWakeUpSchedule
wakeupschedule.isOnline()

Checks if the wake up schedule is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

Returns :

`true` if the wake up schedule can be reached, and `false` otherwise

wakeupschedule→load()wakeupschedule.load() YWakeUpSchedule

Preloads the wake up schedule cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→loadAttribute() YWakeUpSchedule
wakeupschedule.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

wakeupschedule→muteValueCallbacks() YWakeUpSchedule
wakeupschedule.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→nextWakeUpSchedule() YWakeUpSchedule
wakeupschedule.nextWakeUpSchedule()

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

function **nextWakeUpSchedule**( )

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to a wake up schedule currently online, or a `null` pointer if there are no more wake up schedules to enumerate.

wakeupschedule→registerValueCallback() YWakeUpSchedule
wakeupschedule.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.

wakeupschedule→set_hours() YWakeUpSchedule
wakeupschedule→setHours()
wakeupschedule.set_hours()

Changes the hours when a wake up must take place.

function **set_hours**(**newval**)

Parameters :

newval an integer corresponding to the hours when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_logicalName() YWakeUpSchedule
wakeupschedule→setLogicalName()
wakeupschedule.set_logicalName()

Changes the logical name of the wake up schedule.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wake up schedule.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutes() YWakeUpSchedule
wakeupschedule→setMinutes()
wakeupschedule.set_minutes()

Changes all the minutes where a wake up must take place.

function **set_minutes**(**bitmap**)

Parameters :

bitmap Minutes 00-59 of each hour scheduled for wake up.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesA() YWakeUpSchedule
wakeupschedule→setMinutesA()
wakeupschedule.set_minutesA()

Changes the minutes in the 00-29 interval when a wake up must take place.

function **set_minutesA**(**newval**)

Parameters :

newval an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesB() YWakeUpSchedule
wakeupschedule→setMinutesB()
wakeupschedule.set_minutesB()

Changes the minutes in the 30-59 interval when a wake up must take place.

function **set_minutesB**(**newval**)

Parameters :

newval an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_monthDays() YWakeUpSchedule
wakeupschedule→setMonthDays()
wakeupschedule.set_monthDays()

Changes the days of the month when a wake up must take place.

function **set_monthDays**(**newval**)

Parameters :

newval an integer corresponding to the days of the month when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_months() YWakeUpSchedule
wakeupschedule→setMonths()
wakeupschedule.set_months()

Changes the months when a wake up must take place.

function **set_months**(**newval**)

Parameters :

newval an integer corresponding to the months when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_userData() YWakeUpSchedule
wakeupschedule→setUserData()
wakeupschedule.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

wakeupschedule→set_weekDays() YWakeUpSchedule
wakeupschedule→setWeekDays()
wakeupschedule.set_weekDays()

Changes the days of the week when a wake up must take place.

function **set_weekDays**(**newval**)

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→unmuteValueCallbacks() YWakeUpSchedule
wakeupschedule.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→wait_async() YWakeUpSchedule
wakeupschedule.wait_async()

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.72. Watchdog function interface

The watchog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven direcly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_watchdog.js'></script>
cpp	#include "yocto_watchdog.h"
m	#import "yocto_watchdog.h"
pas	uses yocto_watchdog;
vb	yocto_watchdog.vb
cs	yocto_watchdog.cs
java	import com.yoctopuce.YoctoAPI.YWatchdog;
uwp	import com.yoctopuce.YoctoAPI.YWatchdog;
py	from yocto_watchdog import *
php	require_once('yocto_watchdog.php');
es	in HTML: <script src="../../lib/yocto_watchdog.js"></script> in node.js: require('yoctolib-es2017/yocto_watchdog.js');

Global functions

yFindWatchdog(**func**)

Retrieves a watchdog for a given identifier.

yFindWatchdogInContext(**yctx**, **func**)

Retrieves a watchdog for a given identifier in a YAPI context.

yFirstWatchdog()

Starts the enumeration of watchdog currently accessible.

yFirstWatchdogInContext(**yctx**)

Starts the enumeration of watchdog currently accessible.

YWatchdog methods

watchdog→clearCache()

Invalidates the cache.

watchdog→delayedPulse(**ms_delay**, **ms_duration**)

Schedules a pulse.

watchdog→describe()

Returns a short text that describes unambiguously the instance of the watchdog in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

watchdog→get_advertisedValue()

Returns the current value of the watchdog (no more than 6 characters).

watchdog→get_autoStart()

Returns the watchdog runing state at module power on.

watchdog→get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

watchdog→get_errorMessage()

Returns the error message of the latest error with the watchdog.

watchdog→get_errorType()

Returns the numerical error code of the latest error with the watchdog.

watchdog→get_friendlyName()

Returns a global identifier of the watchdog in the format MODULE_NAME.FUNCTION_NAME.

watchdog→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

watchdog→get_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

watchdog→get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.

watchdog→get_logicalName()

Returns the logical name of the watchdog.

watchdog→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

watchdog→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

watchdog→get_module()

Gets the YModule object for the device on which the function is located.

watchdog→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

watchdog→get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

watchdog→get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

watchdog→get_running()

Returns the watchdog running state.

watchdog→get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

watchdog→get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

watchdog→get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

watchdog→get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

watchdog→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

watchdog→isOnline()

Checks if the watchdog is currently reachable, without raising any error.

watchdog→isOnline_async(callback, context)

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

watchdog→load(msValidity)

Preloads the watchdog cache with a specified validity duration.

watchdog→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

watchdog→load_async(msValidity, callback, context)

Preloads the watchdog cache with a specified validity duration (asynchronous version).

watchdog→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

watchdog→nextWatchdog()

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

watchdog→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

watchdog→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

watchdog→resetWatchdog()

Resets the watchdog.

watchdog→set_autoStart(newval)

Changes the watchdog runningsttae at module power on.

watchdog→set_logicalName(newval)

Changes the logical name of the watchdog.

watchdog→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

watchdog→set_maxTimeOnStateB(newval)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

watchdog→set_output(newval)

Changes the output state of the watchdog, when used as a simple switch (single throw).

watchdog→set_running(newval)

Changes the running state of the watchdog.

watchdog→set_state(newval)

Changes the state of the watchdog (A for the idle position, B for the active position).

watchdog→set_stateAtPowerOn(newval)

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

watchdog→set_triggerDelay(newval)

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

watchdog→set_triggerDuration(newval)

Changes the duration of resets caused by the watchdog, in milliseconds.

watchdog→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

watchdog→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

watchdog→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWatchdog.FindWatchdog() YWatchdog yFindWatchdog()yFindWatchdog()

Retrieves a watchdog for a given identifier.

function **FindWatchdog**(**func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWatchdog.isOnline()` to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the watchdog

Returns :

a `YWatchdog` object allowing you to drive the watchdog.

YWatchdog.FindWatchdogInContext() YWatchdog yFindWatchdogInContext()

Retrieves a watchdog for a given identifier in a YAPI context.

function **FindWatchdogInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWatchdog.isOnline()` to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the watchdog

Returns :

a `YWatchdog` object allowing you to drive the watchdog.

YWatchdog.FirstWatchdog() YWatchdog yFirstWatchdog()yFirstWatchdog()

Starts the enumeration of watchdog currently accessible.

function **FirstWatchdog**( )

Use the method `YWatchdog.nextWatchdog()` to iterate on next watchdog.

Returns :

a pointer to a `YWatchdog` object, corresponding to the first watchdog currently online, or a `null` pointer if there are none.

YWatchdog.FirstWatchdogInContext() YWatchdog yFirstWatchdogInContext()

Starts the enumeration of watchdog currently accessible.

function **FirstWatchdogInContext**(**yctx**)

Use the method `YWatchdog.nextWatchdog()` to iterate on next watchdog.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YWatchdog` object, corresponding to the first watchdog currently online, or a `null` pointer if there are none.

watchdog→clearCache()watchdog.clearCache() YWatchdog

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the watchdog attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

watchdog→delayedPulse()watchdog.delayedPulse() YWatchdog

Schedules a pulse.

function **delayedPulse**(**ms_delay**, **ms_duration**)

Parameters :

ms_delay waiting time before the pulse, in millisecondes

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→describe()watchdog.describe() YWatchdog

Returns a short text that describes unambiguously the instance of the watchdog in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the watchdog (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

watchdog→get_advertisedValue() YWatchdog
watchdog→advertisedValue()
watchdog.get_advertisedValue()

Returns the current value of the watchdog (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the watchdog (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

watchdog→get_autoStart() YWatchdog
watchdog→autoStart()watchdog.get_autoStart()

Returns the watchdog runing state at module power on.

function **get_autoStart**( )

Returns :

either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog runing state at module power on

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

watchdog→get_countdown() YWatchdog
watchdog→countdown()watchdog.get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

function **get_countdown**( )

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

watchdog→get_errorMessage() YWatchdog
watchdog→errorMessage()
watchdog.get_errorMessage()

Returns the error message of the latest error with the watchdog.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the watchdog object

watchdog→get_errorType() YWatchdog
watchdog→errorType()watchdog.get_errorType()

Returns the numerical error code of the latest error with the watchdog.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the watchdog object

watchdog→get_friendlyName() YWatchdog
watchdog→friendlyName()
watchdog.get_friendlyName()

Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the watchdog if they are defined, otherwise the serial number of the module and the hardware identifier of the watchdog (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the watchdog using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

watchdog→get_functionDescriptor() YWatchdog
watchdog→functionDescriptor()
watchdog.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

watchdog→get_functionId() YWatchdog
watchdog→functionId()watchdog.get_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the watchdog (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

watchdog→get_hardwareId() YWatchdog
watchdog→hardwareId()watchdog.get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the watchdog (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

watchdog→get_logicalName() YWatchdog
watchdog→logicalName()
watchdog.get_logicalName()

Returns the logical name of the watchdog.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the watchdog.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

watchdog→get_maxTimeOnStateA() YWatchdog
watchdog→maxTimeOnStateA()
watchdog.get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

function **get_maxTimeOnStateA**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

watchdog→get_maxTimeOnStateB() YWatchdog
watchdog→maxTimeOnStateB()
watchdog.get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

function **get_maxTimeOnStateB**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

watchdog→get_module() YWatchdog
watchdog→module()watchdog.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

watchdog→get_output() YWatchdog
watchdog→output()watchdog.get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

function **get_output**( )

Returns :

either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.

watchdog→get_pulseTimer() YWatchdog
watchdog→pulseTimer()watchdog.get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

function **get_pulseTimer**( )

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

watchdog→get_running() YWatchdog
watchdog→running()watchdog.get_running()

Returns the watchdog running state.

function **get_running**( )

Returns :

either Y_RUNNING_OFF or Y_RUNNING_ON, according to the watchdog running state

On failure, throws an exception or returns Y_RUNNING_INVALID.

watchdog→get_state() YWatchdog
watchdog→state()watchdog.get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

function **get_state**( )

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

watchdog→get_stateAtPowerOn() YWatchdog
watchdog→stateAtPowerOn()
watchdog.get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

function **get_stateAtPowerOn**( )

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

watchdog→get_triggerDelay() YWatchdog
watchdog→triggerDelay()
watchdog.get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

function **get_triggerDelay**( )

Returns :

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDELAY_INVALID`.

watchdog→get_triggerDuration() YWatchdog
watchdog→triggerDuration()
watchdog.get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

function **get_triggerDuration**( )

Returns :
an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDURATION_INVALID`.

watchdog→get_userData() YWatchdog
watchdog→userData()watchdog.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.

watchdog→isOnline()watchdog.isOnline() YWatchdog

Checks if the watchdog is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

Returns :

`true` if the watchdog can be reached, and `false` otherwise

watchdog→load()watchdog.load() YWatchdog

Preloads the watchdog cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→loadAttribute()watchdog.loadAttribute() YWatchdog

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

watchdog→muteValueCallbacks() YWatchdog
watchdog.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→nextWatchdog() YWatchdog
watchdog.nextWatchdog()

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

function **nextWatchdog**( )

Returns :

a pointer to a `YWatchdog` object, corresponding to a watchdog currently online, or a `null` pointer if there are no more watchdog to enumerate.

watchdog→pulse()watchdog.pulse() YWatchdog

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

function **pulse**(**ms_duration**)

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→registerValueCallback() YWatchdog
watchdog.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.

watchdog→resetWatchdog() YWatchdog
watchdog.resetWatchdog()

Resets the watchdog.

function **resetWatchdog**( )

When the watchdog is running, this function must be called on a regular basis to prevent the watchog to trigger

Returns :
YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_autoStart() YWatchdog
watchdog→setAutoStart()watchdog.set_autoStart()

Changes the watchdog runningsttae at module power on.

function **set_autoStart**(**newval**)

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the watchdog runningsttae at module power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_logicalName() YWatchdog
watchdog→setLogicalName()
watchdog.set_logicalName()

Changes the logical name of the watchdog.

function **set_logicalName**(**newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the watchdog.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateA() YWatchdog
watchdog→setMaxTimeOnStateA()
watchdog.set_maxTimeOnStateA()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

function **set_maxTimeOnStateA**(**newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateB() YWatchdog
watchdog→setMaxTimeOnStateB()
watchdog.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

function **set_maxTimeOnStateB**(**newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_output() YWatchdog
watchdog→setOutput()watchdog.set_output()

Changes the output state of the watchdog, when used as a simple switch (single throw).

function **set_output**(**newval**)

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_running() YWatchdog
watchdog→setRunning()watchdog.set_running()

Changes the running state of the watchdog.

function **set_running**(**newval**)

Parameters :

newval either `Y_RUNNING_OFF` or `Y_RUNNING_ON`, according to the running state of the watchdog

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_state() YWatchdog
watchdog→setState()watchdog.set_state()

Changes the state of the watchdog (A for the idle position, B for the active position).

function **set_state**(**newval**)

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_stateAtPowerOn() YWatchdog
watchdog→setStateAtPowerOn()
watchdog.set_stateAtPowerOn()

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

function **set_stateAtPowerOn**(**newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDelay() YWatchdog
watchdog→setTriggerDelay()
watchdog.set_triggerDelay()

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

function **set_triggerDelay**(**newval**)

Parameters :

newval an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDuration() YWatchdog
watchdog→setTriggerDuration()
watchdog.set_triggerDuration()

Changes the duration of resets caused by the watchdog, in milliseconds.

function **set_triggerDuration**(**newval**)

Parameters :

newval an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_userData() YWatchdog
watchdog→setUserData()watchdog.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

watchdog→unmuteValueCallbacks() YWatchdog
watchdog.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→wait_async()watchdog.wait_async() YWatchdog

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.73. WeighScale function interface

The YWeighScale class provides a weight measurement from a ratiometric load cell sensor. It can be used to control the bridge excitation parameters, in order to avoid measure shifts caused by temperature variation in the electronics, and can also automatically apply an additional correction factor based on temperature to compensate for offsets in the load cell itself.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_weighscale.js'></script>
cpp	#include "yocto_weighscale.h"
m	#import "yocto_weighscale.h"
pas	uses yocto_weighscale;
vb	yocto_weighscale.vb
cs	yocto_weighscale.cs
java	import com.yoctopuce.YoctoAPI.YWeighScale;
uwp	import com.yoctopuce.YoctoAPI.YWeighScale;
py	from yocto_weighscale import *
php	require_once('yocto_weighscale.php');
es	in HTML: <script src="../../lib/yocto_weighscale.js"></script> in node.js: require('yoctolib-es2017/yocto_weighscale.js');

Global functions

yFindWeighScale(**func**)

Retrieves a weighing scale sensor for a given identifier.

yFindWeighScaleInContext(**yctx**, **func**)

Retrieves a weighing scale sensor for a given identifier in a YAPI context.

yFirstWeighScale()

Starts the enumeration of weighing scale sensors currently accessible.

yFirstWeighScaleInContext(**yctx**)

Starts the enumeration of weighing scale sensors currently accessible.

`YWeighScale` methods

weighscale→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

weighscale→clearCache()

Invalidates the cache.

weighscale→describe()

Returns a short text that describes unambiguously the instance of the weighing scale sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

weighscale→get_adaptRatio()

Returns the compensation temperature update rate, in percents.

weighscale→get_advertisedValue()

Returns the current value of the weighing scale sensor (no more than 6 characters).

weighscale→get_compTemperature()

Returns the current compensation temperature.

weighscale→get_compensation()

Returns the current current thermal compensation value.

weighscale→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

weighscale→get_currentValue()

Returns the current value of the measure, in the specified unit, as a floating point number.

weighscale→get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

weighscale→get_errorMessage()

Returns the error message of the latest error with the weighing scale sensor.

weighscale→get_errorType()

Returns the numerical error code of the latest error with the weighing scale sensor.

weighscale→get_excitation()

Returns the current load cell bridge excitation method.

weighscale→get_friendlyName()

Returns a global identifier of the weighing scale sensor in the format `MODULE_NAME.FUNCTION_NAME`.

weighscale→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

weighscale→get_functionId()

Returns the hardware identifier of the weighing scale sensor, without reference to the module.

weighscale→get_hardwareId()

Returns the unique hardware identifier of the weighing scale sensor in the form `SERIAL.FUNCTIONID`.

weighscale→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

weighscale→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

weighscale→get_logicalName()

Returns the logical name of the weighing scale sensor.

weighscale→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

weighscale→get_module()

Gets the `YModule` object for the device on which the function is located.

weighscale→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

weighscale→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

weighscale→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

weighscale→get_resolution()

Returns the resolution of the measured values.

weighscale→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

weighscale→get_unit()

Returns the measuring unit for the measure.

weighscale→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

weighscale→get_zeroTracking()

Returns the zero tracking threshold value.

weighscale→isOnline()

Checks if the weighing scale sensor is currently reachable, without raising any error.

weighscale→isOnline_async(**callback**, **context**)

Checks if the weighing scale sensor is currently reachable, without raising any error (asynchronous version).

weighscale→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

weighscale→load(**msValidity**)

Preloads the weighing scale sensor cache with a specified validity duration.

weighscale→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

weighscale→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

weighscale→loadOffsetCompensationTable(**tempValues**, **compValues**)

Retrieves the weight offset thermal compensation table previously configured using the `set_offsetCompensationTable` function.

weighscale→loadSpanCompensationTable(**tempValues**, **compValues**)

Retrieves the weight span thermal compensation table previously configured using the `set_spanCompensationTable` function.

weighscale→load_async(**msValidity**, **callback**, **context**)

Preloads the weighing scale sensor cache with a specified validity duration (asynchronous version).

weighscale→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

weighscale→nextWeighScale()

Continues the enumeration of weighing scale sensors started using `yFirstWeighScale()`.

weighscale→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

weighscale→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

weighscale→set_adaptRatio(**newval**)

Changes the compensation temperature update rate, in percents.

weighscale→set_excitation(**newval**)

Changes the current load cell bridge excitation method.

weighscale→set_highestValue(**newval**)

Changes the recorded maximal value observed.

weighscale→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

weighscale→set_logicalName(**newval**)

Changes the logical name of the weighing scale sensor.

weighscale→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

weighscale→set_offsetCompensationTable(**tempValues**, **compValues**)

Records a weight offset thermal compensation table, in order to automatically correct the measured weight based on the compensation temperature.

weighscale→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

weighscale→set_resolution(**newval**)

Changes the resolution of the measured physical values.

weighscale→set_spanCompensationTable(**tempValues**, **compValues**)

Records a weight span thermal compensation table, in order to automatically correct the measured weight based on the compensation temperature.

weighscale→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

weighscale→set_zeroTracking(**newval**)

Changes the compensation temperature update rate, in percents.

weighscale→setupSpan(**currWeight**, **maxWeight**)

Configures the load cell span parameters (stored in the corresponding genericSensor) so that the current signal corresponds to the specified reference weight.

weighscale→startDataLogger()

Starts the data logger on the device.

weighscale→stopDataLogger()

Stops the datalogger on the device.

weighscale→tare()

Adapts the load cell signal bias (stored in the corresponding genericSensor) so that the current signal corresponds to a zero weight.

weighscale→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

weighscale→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWeighScale.FindWeighScale() YWeighScale yFindWeighScale()yFindWeighScale()

Retrieves a weighing scale sensor for a given identifier.

```
function FindWeighScale( 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 weighing scale sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWeighScale.isOnline()` to test if the weighing scale sensor is indeed online at a given time. In case of ambiguity when looking for a weighing scale 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. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the weighing scale sensor

Returns :

a `YWeighScale` object allowing you to drive the weighing scale sensor.

YWeighScale.FindWeighScaleInContext() YWeighScale yFindWeighScaleInContext()

Retrieves a weighing scale sensor for a given identifier in a YAPI context.

function **FindWeighScaleInContext**(**yctx**, **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 weighing scale sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWeighScale.isOnline()` to test if the weighing scale sensor is indeed online at a given time. In case of ambiguity when looking for a weighing scale 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 :

yctx a YAPI context

func a string that uniquely characterizes the weighing scale sensor

Returns :

a `YWeighScale` object allowing you to drive the weighing scale sensor.

YWeighScale.FirstWeighScale() YWeighScale yFirstWeighScale()yFirstWeighScale()

Starts the enumeration of weighing scale sensors currently accessible.

function **FirstWeighScale**( )

Use the method `YWeighScale.nextWeighScale()` to iterate on next weighing scale sensors.

Returns :

a pointer to a `YWeighScale` object, corresponding to the first weighing scale sensor currently online, or a `null` pointer if there are none.

YWeighScale.FirstWeighScaleInContext() YWeighScale yFirstWeighScaleInContext()

Starts the enumeration of weighing scale sensors currently accessible.

function **FirstWeighScaleInContext**(**yctx**)

Use the method `YWeighScale.nextWeighScale()` to iterate on next weighing scale sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YWeighScale` object, corresponding to the first weighing scale sensor currently online, or a `null` pointer if there are none.

weighscale→calibrateFromPoints() YWeighScale
weighscale.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.

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.

weighscale→clearCache()weighscale.clearCache() YWeighScale

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the weighing scale sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

weighscale→describe()weighscale.describe() YWeighScale

Returns a short text that describes unambiguously the instance of the weighing scale sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the weighing scale sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

weighscale→get_adaptRatio() YWeighScale
weighscale→adaptRatio()
weighscale.get_adaptRatio()

Returns the compensation temperature update rate, in percents.

function **get_adaptRatio**( )

the maximal value is 65 percents.

Returns :
a floating point number corresponding to the compensation temperature update rate, in percents

On failure, throws an exception or returns `Y_ADAPTRATIO_INVALID`.

weighscale→get_advertisedValue() YWeighScale
weighscale→advertisedValue()
weighscale.get_advertisedValue()

Returns the current value of the weighing scale sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the weighing scale sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

weighscale→get_compTemperature() YWeighScale
weighscale→compTemperature()
weighscale.get_compTemperature()

Returns the current compensation temperature.

function **get_compTemperature**( )

Returns :
a floating point number corresponding to the current compensation temperature

On failure, throws an exception or returns `Y_COMPTEMPERATURE_INVALID`.

weighscale→get_compensation() YWeighScale
weighscale→compensation()
weighscale.get_compensation()

Returns the current current thermal compensation value.

function **get_compensation**( )

Returns :
a floating point number corresponding to the current current thermal compensation value

On failure, throws an exception or returns `Y_COMPENSATION_INVALID`.

weighscale→get_currentRawValue() YWeighScale
weighscale→currentRawValue()
weighscale.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

function **get_currentRawValue**( )

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

weighscale→get_currentValue() YWeighScale
weighscale→currentValue()
weighscale.get_currentValue()

Returns the current value of the measure, in the specified unit, as a floating point number.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current value of the measure, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

weighscale→get_dataLogger() YWeighScale
weighscale→dataLogger()
weighscale.get_dataLogger()

Returns the YDatalogger object of the device hosting the sensor.

function **get_dataLogger**( )

This method returns an object of class YDatalogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an YDataLogger object or null on error.

weighscale→get_errorMessage() YWeighScale
weighscale→errorMessage()
weighscale.get_errorMessage()

Returns the error message of the latest error with the weighing scale sensor.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the weighing scale sensor object

weighscale→get_errorType() YWeighScale
weighscale→errorType()weighscale.get_errorType()

Returns the numerical error code of the latest error with the weighing scale sensor.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the weighing scale sensor object

weighscale→get_excitation() YWeighScale
weighscale→excitation()weighscale.get_excitation()

Returns the current load cell bridge excitation method.

function **get_excitation**( )

Returns :

a value among `Y_EXCITATION_OFF`, `Y_EXCITATION_DC` and `Y_EXCITATION_AC` corresponding to the current load cell bridge excitation method

On failure, throws an exception or returns `Y_EXCITATION_INVALID`.

weighscale→get_friendlyName() YWeighScale
weighscale→friendlyName()
weighscale.get_friendlyName()

Returns a global identifier of the weighing scale sensor in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the weighing scale sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the weighing scale sensor (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the weighing scale sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

weighscale→get_functionDescriptor() YWeighScale
weighscale→functionDescriptor()
weighscale.get_functionDescriptor()

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.

weighscale→get_functionId() YWeighScale
weighscale→functionId()weighscale.get_functionId()

Returns the hardware identifier of the weighing scale sensor, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the weighing scale sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

weighscale→get_hardwareId() YWeighScale
weighscale→hardwareId()
weighscale.get_hardwareId()

Returns the unique hardware identifier of the weighing scale sensor in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the weighing scale sensor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the weighing scale sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

weighscale→get_highestValue() YWeighScale
weighscale→highestValue()
weighscale.get_highestValue()

Returns the maximal value observed for the measure since the device was started.

function **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

weighscale→get_logFrequency() YWeighScale
weighscale→logFrequency()
weighscale.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

weighscale→get_logicalName() YWeighScale
weighscale→logicalName()
weighscale.get_logicalName()

Returns the logical name of the weighing scale sensor.

function **get_logicalName**( )

Returns :
a string corresponding to the logical name of the weighing scale sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

weighscale→get_lowestValue() YWeighScale
weighscale→lowestValue()
weighscale.get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

weighscale→get_module() YWeighScale
weighscale→module()weighscale.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

weighscale→get_recordedData() YWeighScale
weighscale→recordedData()
weighscale.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

function **get_recordedData**(**startTime**, **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

weighscale→get_reportFrequency() YWeighScale
weighscale→reportFrequency()
weighscale.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

function **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

weighscale→get_resolution() YWeighScale
weighscale→resolution()weighscale.get_resolution()

Returns the resolution of the measured values.

function **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

weighscale→get_sensorState()
weighscale→sensorState()
weighscale.get_sensorState()

YWeighScale

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

function **get_sensorState**( )

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

weighscale→get_unit() YWeighScale
weighscale→unit()weighscale.get_unit()

Returns the measuring unit for the measure.

function **get_unit**( )

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

weighscale→get_userData() YWeighScale
weighscale→userData()weighscale.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.

weighscale→get_zeroTracking() YWeighScale
weighscale→zeroTracking()
weighscale.get_zeroTracking()

Returns the zero tracking threshold value.

function **get_zeroTracking**( )

When this threshold is larger than zero, any measure under the threshold will automatically be ignored and the zero compensation will be updated.

Returns :
a floating point number corresponding to the zero tracking threshold value

On failure, throws an exception or returns `Y_ZEROTRACKING_INVALID`.

weighscale→isOnline()weighscale.isOnline() YWeighScale

Checks if the weighing scale sensor is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the weighing scale sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the weighing scale sensor.

Returns :
`true` if the weighing scale sensor can be reached, and `false` otherwise

weighscale→load()weighscale.load() YWeighScale

Preloads the weighing scale sensor cache with a specified validity duration.

function **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network 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.

weighscale→loadAttribute() YWeighScale
weighscale.loadAttribute()

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

function **loadAttribute**(**attrName**)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

weighscale→loadCalibrationPoints() YWeighScale
weighscale.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

function **loadCalibrationPoints**(**rawValues**, **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→loadOffsetCompensationTable() YWeighScale
weighscale.loadOffsetCompensationTable()

Retrieves the weight offset thermal compensation table previously configured using the `set_offsetCompensationTable` function.

function **loadOffsetCompensationTable**(**tempValues**, **compValues**)

The weight correction is applied by linear interpolation between specified points.

Parameters :

tempValues array of floating point numbers, that is filled by the function with all temperatures for which an offset correction is specified.

compValues array of floating point numbers, that is filled by the function with the offset correction applied for each of the temperature included in the first argument, index by index.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→loadSpanCompensationTable() YWeighScale
weighscale.loadSpanCompensationTable()

Retrieves the weight span thermal compensation table previously configured using the `set_spanCompensationTable` function.

function **loadSpanCompensationTable**(**tempValues**, **compValues**)

The weight correction is applied by linear interpolation between specified points.

Parameters :

tempValues array of floating point numbers, that is filled by the function with all temperatures for which an span correction is specified.

compValues array of floating point numbers, that is filled by the function with the span correction applied for each of the temperature included in the first argument, index by index.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→muteValueCallbacks() YWeighScale
weighscale.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→nextWeighScale() YWeighScale
weighscale.nextWeighScale()

Continues the enumeration of weighing scale sensors started using `yFirstWeighScale()`.

function **nextWeighScale**( )

Returns :

a pointer to a `YWeighScale` object, corresponding to a weighing scale sensor currently online, or a `null` pointer if there are no more weighing scale sensors to enumerate.

weighscale→registerTimedReportCallback() YWeighScale
weighscale.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

function **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

weighscale→registerValueCallback() YWeighScale
weighscale.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.

weighscale→set_adaptRatio() YWeighScale
weighscale→setAdaptRatio()
weighscale.set_adaptRatio()

Changes the compensation temperature update rate, in percents.

function **set_adaptRatio**(**newval**)

Parameters :

newval a floating point number corresponding to the compensation temperature update rate, in percents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_excitation()
weighscale→setExcitation()
weighscale.set_excitation()

YWeighScale

Changes the current load cell bridge excitation method.

function **set_excitation**(**newval**)

Parameters :

newval a value among `Y_EXCITATION_OFF`, `Y_EXCITATION_DC` and `Y_EXCITATION_AC` corresponding to the current load cell bridge excitation method

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_highestValue() YWeighScale
weighscale→setHighestValue()
weighscale.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.

weighscale→set_logFrequency() YWeighScale
weighscale→setLogFrequency()
weighscale.set_logFrequency()

Changes the datalogger recording frequency for this function.

function **set_logFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_logicalName() YWeighScale
weighscale→setLogicalName()
weighscale.set_logicalName()

Changes the logical name of the weighing scale 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 weighing scale sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_lowestValue() YWeighScale
weighscale→setLowestValue()
weighscale.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.

weighscale→set_offsetCompensationTable() YWeighScale
weighscale→setOffsetCompensationTable()
weighscale.set_offsetCompensationTable()

Records a weight offset thermal compensation table, in order to automatically correct the measured weight based on the compensation temperature.

function **set_offsetCompensationTable**(**tempValues**, **compValues**)

The weight correction will be applied by linear interpolation between specified points.

Parameters :

tempValues array of floating point numbers, corresponding to all temperatures for which an offset correction is specified.

compValues array of floating point numbers, corresponding to the offset correction to apply for each of the temperature included in the first argument, index by index.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_reportFrequency() YWeighScale
weighscale→setReportFrequency()
weighscale.set_reportFrequency()

Changes the timed value notification frequency for this function.

function **set_reportFrequency**(**newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_resolution() YWeighScale
weighscale→setResolution()
weighscale.set_resolution()

Changes the resolution of the measured physical values.

function **set_resolution**(**newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_spanCompensationTable() YWeighScale
weighscale→setSpanCompensationTable()
weighscale.set_spanCompensationTable()

Records a weight span thermal compensation table, in order to automatically correct the measured weight based on the compensation temperature.

function **set_spanCompensationTable**(**tempValues**, **compValues**)

The weight correction will be applied by linear interpolation between specified points.

Parameters :

tempValues array of floating point numbers, corresponding to all temperatures for which a span correction is specified.

compValues array of floating point numbers, corresponding to the span correction (in percents) to apply for each of the temperature included in the first argument, index by index.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→set_userData()
weighscale→setUserData()
weighscale.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

weighscale→set_zeroTracking() YWeighScale
weighscale→setZeroTracking()
weighscale.set_zeroTracking()

Changes the compensation temperature update rate, in percents.

function **set_zeroTracking**(**newval**)

Parameters :

newval a floating point number corresponding to the compensation temperature update rate, in percents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→setupSpan()weighscale.setupSpan() YWeighScale

Configures the load cell span parameters (stored in the corresponding genericSensor) so that the current signal corresponds to the specified reference weight.

function **setupSpan**(**currWeight**, **maxWeight**)

Parameters :

currWeight reference weight presently on the load cell.

maxWeight maximum weight to be expectect on the load cell.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→startDataLogger() YWeighScale
weighscale.startDataLogger()

Starts the data logger on the device.

```
function startDataLogger( )
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
YAPI_SUCCESS if the call succeeds.

weighscale→stopDataLogger() YWeighScale
weighscale.stopDataLogger()

Stops the datalogger on the device.

function **stopDataLogger**( )

Returns :

YAPI_SUCCESS if the call succeeds.

weighscale→tare()weighscale.tare() YWeighScale

Adapts the load cell signal bias (stored in the corresponding genericSensor) so that the current signal corresponds to a zero weight.

function **tare**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→unmuteValueCallbacks() YWeighScale
weighscale.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

weighscale→wait_async()weighscale.wait_async() YWeighScale

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.74. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_wireless.js'></script>
cpp     #include "yocto_wireless.h"
m       #import "yocto_wireless.h"
pas     uses yocto_wireless;
vb      yocto_wireless.vb
cs      yocto_wireless.cs
java    import com.yoctopuce.YoctoAPI.YWireless;
uwp     import com.yoctopuce.YoctoAPI.YWireless;
py      from yocto_wireless import *
php     require_once('yocto_wireless.php');
es      in HTML: <script src="../../lib/yocto_wireless.js"></script>
        in node.js: require('yoctolib-es2017/yocto_wireless.js');
```

Global functions

yFindWireless(**func**)

Retrieves a wireless lan interface for a given identifier.

yFindWirelessInContext(**yctx**, **func**)

Retrieves a wireless lan interface for a given identifier in a YAPI context.

yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

yFirstWirelessInContext(**yctx**)

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→clearCache()

Invalidates the cache.

wireless→describe()

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wireless→get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

wireless→get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

wireless→get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

wireless→get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

wireless→get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

wireless→get_friendlyName()

Returns a global identifier of the wireless lan interface in the format MODULE_NAME.FUNCTION_NAME.

wireless→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

wireless→get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

wireless→get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form SERIAL.FUNCTIONID.

wireless→get_linkQuality()

Returns the link quality, expressed in percent.

wireless→get_logicalName()

Returns the logical name of the wireless lan interface.

wireless→get_message()

Returns the latest status message from the wireless interface.

wireless→get_module()

Gets the YModule object for the device on which the function is located.

wireless→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

wireless→get_security()

Returns the security algorithm used by the selected wireless network.

wireless→get_ssid()

Returns the wireless network name (SSID).

wireless→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

wireless→get_wlanState()

Returns the current state of the wireless interface.

wireless→isOnline()

Checks if the wireless lan interface is currently reachable, without raising any error.

wireless→isOnline_async(**callback**, **context**)

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

wireless→joinNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

wireless→load(**msValidity**)

Preloads the wireless lan interface cache with a specified validity duration.

wireless→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

wireless→load_async(**msValidity**, **callback**, **context**)

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

wireless→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

wireless→nextWireless()

Continues the enumeration of wireless lan interfaces started using yFirstWireless().

wireless→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

wireless→set_logicalName(newval)

Changes the logical name of the wireless lan interface.

wireless→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

wireless→softAPNetwork(ssid, securityKey)

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

wireless→startWlanScan()

Triggers a scan of the wireless frequency and builds the list of available networks.

wireless→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

wireless→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWireless.FindWireless() YWireless yFindWireless()yFindWireless()

Retrieves a wireless lan interface for a given identifier.

```
function FindWireless( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wireless lan interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWireless.isOnline()` to test if the wireless lan interface is indeed online at a given time. In case of ambiguity when looking for a wireless lan interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name. If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the wireless lan interface

Returns :

a `YWireless` object allowing you to drive the wireless lan interface.

YWireless.FindWirelessInContext() YWireless
yFindWirelessInContext()

Retrieves a wireless lan interface for a given identifier in a YAPI context.

function **FindWirelessInContext**(**yctx**, **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 :

yctx a YAPI context

func a string that uniquely characterizes the wireless lan interface

Returns :

a `YWireless` object allowing you to drive the wireless lan interface.

YWireless.FirstWireless() YWireless yFirstWireless()yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

```
function FirstWireless( )
```

Use the method `YWireless.nextWireless()` to iterate on next wireless lan interfaces.

Returns :

a pointer to a `YWireless` object, corresponding to the first wireless lan interface currently online, or a `null` pointer if there are none.

YWireless.FirstWirelessInContext() YWireless
yFirstWirelessInContext()

Starts the enumeration of wireless lan interfaces currently accessible.

function **FirstWirelessInContext**(**yctx**)

Use the method `YWireless.nextWireless()` to iterate on next wireless lan interfaces.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YWireless` object, corresponding to the first wireless lan interface currently online, or a `null` pointer if there are none.

wireless→adhocNetwork()wireless.adhocNetwork() YWireless

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

function **adhocNetwork**(**ssid**, **securityKey**)

On the YoctoHub-Wireless-g, it is best to use softAPNetworkInstead(), which emulates an access point (Soft AP) which is more efficient and more widely supported than ad-hoc networks.

When a security key is specified for an ad-hoc network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→clearCache()wireless.clearCache() YWireless

Invalidates the cache.

function **clearCache**( )

Invalidates the cache of the wireless lan interface attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

wireless→describe()wireless.describe() YWireless

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form TYPE(NAME)=SERIAL.FUNCTIONID.

function **describe**( )

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the wireless lan interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

wireless→get_advertisedValue() YWireless
wireless→advertisedValue()
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() YWireless
wireless→channel()wireless.get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

function **get_channel**( )

Returns :

an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns `Y_CHANNEL_INVALID`.

wireless→get_detectedWlans() YWireless
wireless→detectedWlans()
wireless.get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

function **get_detectedWlans**( )

This list is not updated when the module is already connected to an acces point (infrastructure mode). To force an update of this list, `startWlanScan()` must be called. Note that an languages without garbage collections, the returned list must be freed by the caller.

Returns :

a list of `YWlanRecord` objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.

wireless→get_errorMessage() YWireless
wireless→errorMessage()
wireless.get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

function **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the wireless lan interface object

wireless→get_errorType() YWireless
wireless→errorType()wireless.get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

function **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the wireless lan interface object

wireless→get_friendlyName() YWireless
wireless→friendlyName()wireless.get_friendlyName()

Returns a global identifier of the wireless lan interface in the format MODULE_NAME.FUNCTION_NAME.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the wireless lan interface if they are defined, otherwise the serial number of the module and the hardware identifier of the wireless lan interface (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the wireless lan interface using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

wireless→get_functionDescriptor() YWireless
wireless→functionDescriptor()
wireless.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wireless→get_functionId() YWireless
wireless→functionId()wireless.get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

function **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the wireless lan interface (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wireless→get_hardwareId() YWireless
wireless→hardwareId()wireless.get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form SERIAL.FUNCTIONID.

function **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wireless lan interface (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the wireless lan interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wireless→get_linkQuality() YWireless
wireless→linkQuality()wireless.get_linkQuality()

Returns the link quality, expressed in percent.

function **get_linkQuality**( )

Returns :
an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns `Y_LINKQUALITY_INVALID`.

wireless→get_logicalName() YWireless
wireless→logicalName()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_message() YWireless
wireless→message()wireless.get_message()

Returns the latest status message from the wireless interface.

function **get_message**( )

Returns :
a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns `Y_MESSAGE_INVALID`.

wireless→get_module() YWireless
wireless→module()wireless.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

wireless→get_security() YWireless
wireless→security()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() YWireless
wireless→ssid()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() YWireless
wireless→userData()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→get_wlanState() YWireless
wireless→wlanState()wireless.get_wlanState()

Returns the current state of the wireless interface.

function **get_wlanState**( )

The state Y_WLANSTATE_DOWN means that the network interface is not connected to a network. The state Y_WLANSTATE_SCANNING means that the network interface is scanning available frequencies. During this stage, the device is not reachable, and the network settings are not yet applied. The state Y_WLANSTATE_CONNECTED means that the network settings have been successfully applied ant that the device is reachable from the wireless network. If the device is configured to use ad-hoc or Soft AP mode, it means that the wireless network is up and that other devices can join the network. The state Y_WLANSTATE_REJECTED means that the network interface has not been able to join the requested network. The description of the error can be obtain with the get_message() method.

Returns :

a value among Y_WLANSTATE_DOWN, Y_WLANSTATE_SCANNING, Y_WLANSTATE_CONNECTED and Y_WLANSTATE_REJECTED corresponding to the current state of the wireless interface

On failure, throws an exception or returns Y_WLANSTATE_INVALID.

wireless→isOnline()wireless.isOnline() YWireless

Checks if the wireless lan interface is currently reachable, without raising any error.

function **isOnline**( )

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

Returns :

`true` if the wireless lan interface can be reached, and `false` otherwise

wireless→joinNetwork()wireless.joinNetwork() YWireless

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

function **joinNetwork**(**ssid**, **securityKey**)

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→load()wireless.load() YWireless

Preloads the wireless lan interface cache with a specified validity duration.

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→loadAttribute()wireless.loadAttribute() YWireless

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
function loadAttribute( attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

wireless→muteValueCallbacks() YWireless
wireless.muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

function **muteValueCallbacks**( )

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→nextWireless()wireless.nextWireless() YWireless

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

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() YWireless
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() YWireless
wireless→setLogicalName()
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() YWireless
wireless→setUserData()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

wireless→softAPNetwork()wireless.softAPNetwork() YWireless

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

function **softAPNetwork**(**ssid**, **securityKey**)

This function can only be used with the YoctoHub-Wireless-g.

When a security key is specified for a SoftAP network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→startWlanScan()wireless.startWlanScan() YWireless

Triggers a scan of the wireless frequency and builds the list of available networks.

function **startWlanScan**( )

The scan forces a disconnection from the current network. At then end of the process, the the network interface attempts to reconnect to the previous network. During the scan, the `wlanState` switches to `Y_WLANSTATE_DOWN`, then to `Y_WLANSTATE_SCANNING`. When the scan is completed, `get_wlanState()` returns either `Y_WLANSTATE_DOWN` or `Y_WLANSTATE_SCANNING`. At this point, the list of detected network can be retrieved with the `get_detectedWlans()` method.

On failure, throws an exception or returns a negative error code.

wireless→unmuteValueCallbacks() YWireless
wireless.unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

function **unmuteValueCallbacks**( )

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→wait_async()wireless.wait_async() YWireless

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z