# Objective-C API Reference

# Table of contents

# 1. Introduction

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

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

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

# 2. Using Yocto-Demo with Objective-C

Objective-C is language of choice for programming on Mac OS X, due to its integration with the Cocoa framework. In order to use the Objective-C library, you need XCode version 4.2 (earlier versions will not work), available freely when you run Lion. If you are still under Snow Leopard, you need to be registered as Apple developer to be able to download XCode 4.2. The Yoctopuce library is ARC compatible. You can therefore implement your projects either using the traditional *retain / release* method, or using the *Automatic Reference Counting*.

Yoctopuce Objective-C libraries[1] are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from Objective-C.

You will soon notice that the Objective-C API defines many functions which return objects. You do not need to deallocate these objects yourself, the API does it automatically at the end of the application.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface. You can find on Yoctopuce blog a detailed example[2] with video shots showing how to integrate the library into your projects.

## 2.1. Control of the Led function

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/ Doc-GettingStarted-Yocto-Demo** of the Yoctopuce library.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"
#import "yocto_led.h"


static void usage(void)
{
    NSLog(@"usage: demo <serial_number>  [ on | off ]");
    NSLog(@"       demo <logical_name> [ on | off ]");
    NSLog(@"       demo any [ on | off ]                (use any discovered device)");
    exit(1);
}
```

[1] www.yoctopuce.com/EN/libraries.php
[2] www.yoctopuce.com/EN/article/new-objective-c-library-for-mac-os-x

```
int main(int argc, const char * argv[])
{
    NSError *error;
    if(argc < 3) {
        usage();
    }

    @autoreleasepool {
        NSString *target = [NSString stringWithUTF8String:argv[1]];
        NSString *on_off = [NSString stringWithUTF8String:argv[2]];
        YLed     *led;

        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if([target isEqualToString:@"any"]){
            led =  [YLed FirstLed];
        }else{
            led =  [YLed FindLed:[target stringByAppendingString:@".led"]];
        }
        if ([led isOnline]) {
            if ([on_off isEqualToString:@"on"])
                [led set_power:Y_POWER_ON];
            else
                [led set_power:Y_POWER_OFF];
        } else {
            NSLog(@"Module not connected (check identification and USB cable)\n");
        }
    }
    return 0;
}
```

There are only a few really important lines in this example. We will look at them in details.

### yocto_api.h et yocto_led.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_led.h` is necessary to manage modules containing a led, such as Yocto-Demo.

### yRegisterHub

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

### yFindLed

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

```
YLed *led = yFindLed(@"YCTOPOC1-123456.led");
YLed *led = yFindLed(@"YCTOPOC1-123456.MyFunction");
YLed *led = yFindLed(@"MyModule.led");
YLed *led = yFindLed(@"MyModule.MyFunction");
YLed *led = yFindLed(@"MyFunction");
```

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

### isOnline

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

### set_power

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

## 2.2. 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.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

static void usage(const char *exe)
{
    NSLog(@"usage: %s <serial or logical name> [ON/OFF]\n",exe);
    exit(1);
}


int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if(argc < 2)
            usage(argv[0]);
        NSString *serial_or_name =[NSString stringWithUTF8String:argv[1]];
        YModule *module = [YModule FindModule:serial_or_name];  // use serial or logical
name
        if ([module isOnline]) {
            if (argc > 2) {
                if (strcmp(argv[2], "ON")==0)
                    [module setBeacon:Y_BEACON_ON];
                else
                    [module setBeacon:Y_BEACON_OFF];
            }
            NSLog(@"serial:       %@\n", [module serialNumber]);
            NSLog(@"logical name: %@\n", [module logicalName]);
            NSLog(@"luminosity:   %d\n", [module luminosity]);
            NSLog(@"beacon:       ");
            if ([module beacon] == Y_BEACON_ON)
               NSLog(@"ON\n");
            else
               NSLog(@"OFF\n");
            NSLog(@"upTime:       %ld sec\n", [module upTime]/1000);
            NSLog(@"USB current:  %d mA\n",  [module usbCurrent]);
            NSLog(@"logs:  %@\n",  [module get_lastLogs]);
        } else {
            NSLog(@"%@ not connected (check identification and USB cable)\n",serial_or_name
);
        }
    }
    return 0;
}
```

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.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

static void usage(const char *exe)
{
    NSLog(@"usage: %s <serial> <newLogicalName>\n",exe);
    exit(1);
}


int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb" :&error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }

        if(argc < 2)
            usage(argv[0]);

        NSString *serial_or_name =[NSString stringWithUTF8String:argv[1]];
        YModule *module = [YModule FindModule:serial_or_name];  // use serial or logical
name

        if (module.isOnline) {
            if (argc >= 3){
                NSString *newname =  [NSString stringWithUTF8String:argv[2]];
                if (![YAPI CheckLogicalName:newname]){
                    NSLog(@"Invalid name (%@)\n", newname);
                    usage(argv[0]);
                }
                module.logicalName = newname;
                [module saveToFlash];
            }
            NSLog(@"Current name: %@\n", module.logicalName);
        } else {
            NSLog(@"%@ not connected (check identification and USB cable)\n",serial_or_name
);
        }
    }
    return 0;
}
```

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

## Listing the modules

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

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

int main (int argc, const char * argv[])
{
    NSError *error;
```

```
    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb" :&error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@\n", [error localizedDescription]);
            return 1;
        }

        NSLog(@"Device list:\n");

        YModule *module = [YModule FirstModule];
        while (module != nil) {
            NSLog(@"%@ %@",module.serialNumber, module.productName);
            module = [module nextModule];
        }
    }
    return 0;
}
```

## 2.3. Error handling

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

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

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

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

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

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

# 3. Reference

## 3.1. General functions

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

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

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

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

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

**yRegisterLogFunction(logfun)**

Registers a log callback function.

**ySelectArchitecture(arch)**

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

**ySetDelegate(object)**

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

**ySetTimeout(callback, ms_timeout, arguments)**

Invoke the specified callback function after a given timeout.

**ySleep(ms_duration, errmsg)**

Pauses the execution flow for a specified duration.

**yTriggerHubDiscovery(errmsg)**

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

**yUnregisterHub(url)**

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

**yUpdateDeviceList(errmsg)**

Triggers a (re)detection of connected Yoctopuce modules.

**yUpdateDeviceList_async(callback, context)**

Triggers a (re)detection of connected Yoctopuce modules.

## YAPI.CheckLogicalName() yCheckLogicalName()yCheckLogicalName() — YAPI

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

| | |
|---|---|
| js | function **yCheckLogicalName**( **name**) |
| nodejs | function **CheckLogicalName**( **name**) |
| php | function **yCheckLogicalName**( **$name**) |
| cpp | bool **yCheckLogicalName**( const string& **name**) |
| m | BOOL **yCheckLogicalName**( NSString * **name**) |
| pas | function **yCheckLogicalName**( **name**: string): boolean |
| vb | function **yCheckLogicalName**( ByVal **name** As String) As Boolean |
| cs | bool **CheckLogicalName**( string **name**) |
| java | boolean **CheckLogicalName**( String **name**) |
| py | def **CheckLogicalName**( **name**) |

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

**Parameters :**

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

**Returns :**

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

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

Disables the use of exceptions to report runtime errors.

| | |
|---|---|
| js | function **yDisableExceptions**( ) |
| nodejs | function **DisableExceptions**( ) |
| php | function **yDisableExceptions**( ) |
| cpp | void **yDisableExceptions**( ) |
| m | void **yDisableExceptions**( ) |
| pas | procedure **yDisableExceptions**( ) |
| vb | procedure **yDisableExceptions**( ) |
| cs | void **DisableExceptions**( ) |
| py | def **DisableExceptions**( ) |

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

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

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

| | |
|---|---|
| `js` | function **yEnableExceptions**( ) |
| `nodejs` | function **EnableExceptions**( ) |
| `php` | function **yEnableExceptions**( ) |
| `cpp` | void **yEnableExceptions**( ) |
| `m` | void **yEnableExceptions**( ) |
| `pas` | procedure **yEnableExceptions**( ) |
| `vb` | procedure **yEnableExceptions**( ) |
| `cs` | void **EnableExceptions**( ) |
| `py` | def **EnableExceptions**( ) |

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

## YAPI.EnableUSBHost() yEnableUSBHost() — YAPI

This function is used only on Android.

`java` void **EnableUSBHost**( Object **osContext**)

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

**Parameters :**

**osContext** an object of class android.content.Context (or any subclass).

## YAPI.FreeAPI() yFreeAPI()yFreeAPI() — YAPI

Frees dynamically allocated memory blocks used by the Yoctopuce library.

| | |
|---|---|
| js | function **yFreeAPI**( ) |
| nodejs | function **FreeAPI**( ) |
| php | function **yFreeAPI**( ) |
| cpp | void **yFreeAPI**( ) |
| m | void **yFreeAPI**( ) |
| pas | procedure **yFreeAPI**( ) |
| vb | procedure **yFreeAPI**( ) |
| cs | void **FreeAPI**( ) |
| java | void **FreeAPI**( ) |
| py | def **FreeAPI**( ) |

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

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

Returns the version identifier for the Yoctopuce library in use.

| | |
|---|---|
| js | function **yGetAPIVersion**( ) |
| nodejs | function **GetAPIVersion**( ) |
| php | function **yGetAPIVersion**( ) |
| cpp | string **yGetAPIVersion**( ) |
| m | NSString* **yGetAPIVersion**( ) |
| pas | function **yGetAPIVersion**( ): string |
| vb | function **yGetAPIVersion**( ) As String |
| cs | String **GetAPIVersion**( ) |
| java | String **GetAPIVersion**( ) |
| py | def **GetAPIVersion**( ) |

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

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

**Returns :**

a character string describing the library version.

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

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

| | |
|---|---|
| js | function **yGetTickCount**( ) |
| nodejs | function **GetTickCount**( ) |
| php | function **yGetTickCount**( ) |
| cpp | u64 **yGetTickCount**( ) |
| m | u64 **yGetTickCount**( ) |
| pas | function **yGetTickCount**( ): u64 |
| vb | function **yGetTickCount**( ) As Long |
| cs | ulong **GetTickCount**( ) |
| java | long **GetTickCount**( ) |
| py | def **GetTickCount**( ) |

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

**Returns :**

a long integer corresponding to the millisecond counter.

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

Maintains the device-to-library communication channel.

| | |
|---|---|
| js | function **yHandleEvents**( **errmsg**) |
| nodejs | function **HandleEvents**( **errmsg**) |
| php | function **yHandleEvents**( **&$errmsg**) |
| cpp | YRETCODE **yHandleEvents**( string& **errmsg**) |
| m | YRETCODE **yHandleEvents**( NSError** **errmsg**) |
| pas | function **yHandleEvents**( var **errmsg**: string): integer |
| vb | function **yHandleEvents**( ByRef **errmsg** As String) As YRETCODE |
| cs | YRETCODE **HandleEvents**( ref string **errmsg**) |
| java | int **HandleEvents**( ) |
| py | def **HandleEvents**( **errmsg**=None) |

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

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

**Parameters :**

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

**Returns :**

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

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

Initializes the Yoctopuce programming library explicitly.

| | |
|---|---|
| js | function **yInitAPI**( **mode**, **errmsg**) |
| nodejs | function **InitAPI**( **mode**, **errmsg**) |
| php | function **yInitAPI**( **$mode**, **&$errmsg**) |
| cpp | YRETCODE **yInitAPI**( int **mode**, string& **errmsg**) |
| m | YRETCODE **yInitAPI**( int **mode**, NSError** **errmsg**) |
| pas | function **yInitAPI**( **mode**: integer, var **errmsg**: string): integer |
| vb | function **yInitAPI**( ByVal **mode** As Integer, ByRef **errmsg** As String) As Integer |
| cs | int **InitAPI**( int **mode**, ref string **errmsg**) |
| java | int **InitAPI**( int **mode**) |
| py | def **InitAPI**( **mode**, **errmsg**=None) |

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

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

**Parameters :**

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

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

**Returns :**

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

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

Fault-tolerant alternative to RegisterHub().

| | |
|---|---|
| js | function **yPreregisterHub**( **url**, **errmsg**) |
| nodejs | function **PreregisterHub**( **url**, **errmsg**) |
| php | function **yPreregisterHub**( **$url**, **&$errmsg**) |
| cpp | YRETCODE **yPreregisterHub**( const string& **url**, string& **errmsg**) |
| m | YRETCODE **yPreregisterHub**( NSString * **url**, NSError** **errmsg**) |
| pas | function **yPreregisterHub**( **url**: string, var **errmsg**: string): integer |
| vb | function **yPreregisterHub**( ByVal **url** As String, ByRef **errmsg** As String) As Integer |
| cs | int **PreregisterHub**( string **url**, ref string **errmsg**) |
| java | int **PreregisterHub**( String **url**) |
| py | def **PreregisterHub**( **url**, **errmsg**=None) |

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

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

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

| | |
|---|---|
| js | function **yRegisterDeviceArrivalCallback**( **arrivalCallback**) |
| nodejs | function **RegisterDeviceArrivalCallback**( **arrivalCallback**) |
| php | function **yRegisterDeviceArrivalCallback**( **$arrivalCallback**) |
| cpp | void **yRegisterDeviceArrivalCallback**( yDeviceUpdateCallback **arrivalCallback**) |
| m | void **yRegisterDeviceArrivalCallback**( yDeviceUpdateCallback **arrivalCallback**) |
| pas | procedure **yRegisterDeviceArrivalCallback**( **arrivalCallback**: yDeviceUpdateFunc) |
| vb | procedure **yRegisterDeviceArrivalCallback**( ByVal **arrivalCallback** As yDeviceUpdateFunc) |
| cs | void **RegisterDeviceArrivalCallback**( yDeviceUpdateFunc **arrivalCallback**) |
| java | void **RegisterDeviceArrivalCallback**( DeviceArrivalCallback **arrivalCallback**) |
| py | def **RegisterDeviceArrivalCallback**( **arrivalCallback**) |

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

**Parameters :**

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

## YAPI.RegisterDeviceRemovalCallback() yRegisterDeviceRemovalCallback() yRegisterDeviceRemovalCallback() YAPI

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

| | |
|---|---|
| js | function **yRegisterDeviceRemovalCallback**( **removalCallback**) |
| nodejs | function **RegisterDeviceRemovalCallback**( **removalCallback**) |
| php | function **yRegisterDeviceRemovalCallback**( **$removalCallback**) |
| cpp | void **yRegisterDeviceRemovalCallback**( yDeviceUpdateCallback **removalCallback**) |
| m | void **yRegisterDeviceRemovalCallback**( yDeviceUpdateCallback **removalCallback**) |
| pas | procedure **yRegisterDeviceRemovalCallback**( **removalCallback**: yDeviceUpdateFunc) |
| vb | procedure **yRegisterDeviceRemovalCallback**( ByVal **removalCallback** As yDeviceUpdateFunc) |
| cs | void **RegisterDeviceRemovalCallback**( yDeviceUpdateFunc **removalCallback**) |
| java | void **RegisterDeviceRemovalCallback**( DeviceRemovalCallback **removalCallback**) |
| py | def **RegisterDeviceRemovalCallback**( **removalCallback**) |

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

**Parameters :**

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

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

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

| | |
|---|---|
| js | function **yRegisterHub**( **url**, **errmsg**) |
| nodejs | function **RegisterHub**( **url**, **errmsg**) |
| php | function **yRegisterHub**( **$url**, **&$errmsg**) |
| cpp | YRETCODE **yRegisterHub**( const string& **url**, string& **errmsg**) |
| m | YRETCODE **yRegisterHub**( NSString * **url**, NSError** **errmsg**) |
| pas | function **yRegisterHub**( **url**: string, var **errmsg**: string): integer |
| vb | function **yRegisterHub**( ByVal **url** As String,<br>ByRef **errmsg** As String) As Integer |
| cs | int **RegisterHub**( string **url**, ref string **errmsg**) |
| java | int **RegisterHub**( String **url**) |
| py | def **RegisterHub**( **url**, **errmsg**=None) |

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

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

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

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

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

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

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

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

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

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

| | |
|---|---|
| cpp | void **yRegisterHubDiscoveryCallback**( YHubDiscoveryCallback **hubDiscoveryCallback**) |
| m | +(void) **yRegisterHubDiscoveryCallback** : (YHubDiscoveryCallback) **hubDiscoveryCallback** |
| pas | procedure **yRegisterHubDiscoveryCallback**( **hubDiscoveryCallback**: YHubDiscoveryCallback) |
| vb | procedure **yRegisterHubDiscoveryCallback**( ByVal **hubDiscoveryCallback** As YHubDiscoveryCallback) |
| cs | void **RegisterHubDiscoveryCallback**( YHubDiscoveryCallback **hubDiscoveryCallback**) |
| java | void **RegisterHubDiscoveryCallback**( HubDiscoveryCallback **hubDiscoveryCallback**) |
| py | def **RegisterHubDiscoveryCallback**( **hubDiscoveryCallback**) |

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

**Parameters :**

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

## YAPI.RegisterLogFunction() yRegisterLogFunction()yRegisterLogFunction() YAPI

Registers a log callback function.

| | |
|---|---|
| cpp | void **yRegisterLogFunction**( yLogFunction **logfun**) |
| m | void **yRegisterLogFunction**( yLogCallback **logfun**) |
| pas | procedure **yRegisterLogFunction**( **logfun**: yLogFunc) |
| vb | procedure **yRegisterLogFunction**( ByVal **logfun** As yLogFunc) |
| cs | void **RegisterLogFunction**( yLogFunc **logfun**) |
| java | void **RegisterLogFunction**( LogCallback **logfun**) |
| py | def **RegisterLogFunction**( **logfun**) |

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

**Parameters :**

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

## YAPI.SelectArchitecture() ySelectArchitecture() YAPI

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

```
py def SelectArchitecture( arch)
```

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

**Parameters :**

**arch** A string containing the architecture to use. Possibles value are: `"armhf"`,`"armel"`, `"i386"`,`"x86_64"`,`"32bit"`, `"64bit"`

**Returns :**

nothing.

On failure, throws an exception.

## YAPI.SetDelegate() ySetDelegate()ySetDelegate()

**YAPI**

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

| m | void **ySetDelegate**( id **object**) |
|---|---|

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

**Parameters :**

**object** an object that must follow the protocol `YAPIDelegate`, or `nil`

## YAPI.SetTimeout() ySetTimeout()

**YAPI**

Invoke the specified callback function after a given timeout.

`js` function **ySetTimeout**( **callback**, **ms_timeout**, **arguments**)
`nodejs` function **SetTimeout**( **callback**, **ms_timeout**, **arguments**)

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

**Parameters :**

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

**Returns :**

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

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

**YAPI**

Pauses the execution flow for a specified duration.

| | |
|---|---|
| js | function **ySleep**( **ms_duration**, **errmsg**) |
| nodejs | function **Sleep**( **ms_duration**, **errmsg**) |
| php | function **ySleep**( **$ms_duration**, **&$errmsg**) |
| cpp | YRETCODE **ySleep**( unsigned **ms_duration**, string& **errmsg**) |
| m | YRETCODE **ySleep**( unsigned **ms_duration**, NSError ** **errmsg**) |
| pas | function **ySleep**( **ms_duration**: integer, var **errmsg**: string): integer |
| vb | function **ySleep**( ByVal **ms_duration** As Integer, ByRef **errmsg** As String) As Integer |
| cs | int **Sleep**( int **ms_duration**, ref string **errmsg**) |
| java | int **Sleep**( long **ms_duration**) |
| py | def **Sleep**( **ms_duration**, **errmsg**=None) |

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

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

**Parameters :**

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

**Returns :**

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

## YAPI.TriggerHubDiscovery() yTriggerHubDiscovery()yTriggerHubDiscovery() YAPI

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

| | |
|---|---|
| cpp | YRETCODE **yTriggerHubDiscovery**( string& **errmsg**) |
| m | +(YRETCODE) **yTriggerHubDiscovery** : (NSError**) **errmsg** |
| pas | function **yTriggerHubDiscovery**( var **errmsg**: string): integer |
| vb | function **yTriggerHubDiscovery**( ByRef **errmsg** As String) As Integer |
| cs | int **TriggerHubDiscovery**( ref string **errmsg**) |
| java | int **TriggerHubDiscovery**( ) |
| py | def **TriggerHubDiscovery**( **errmsg**=None) |

**Parameters :**

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

**Returns :**

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

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

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

| | |
|---|---|
| js | function **yUnregisterHub**( **url**) |
| nodejs | function **UnregisterHub**( **url**) |
| php | function **yUnregisterHub**( **$url**) |
| cpp | void **yUnregisterHub**( const string& **url**) |
| m | void **yUnregisterHub**( NSString * **url**) |
| pas | procedure **yUnregisterHub**( **url**: string) |
| vb | procedure **yUnregisterHub**( ByVal **url** As String) |
| cs | void **UnregisterHub**( string **url**) |
| java | void **UnregisterHub**( String **url**) |
| py | def **UnregisterHub**( **url**) |

**Parameters :**

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

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

Triggers a (re)detection of connected Yoctopuce modules.

| | |
|---|---|
| js | function **yUpdateDeviceList**( **errmsg**) |
| nodejs | function **UpdateDeviceList**( **errmsg**) |
| php | function **yUpdateDeviceList**( **&$errmsg**) |
| cpp | YRETCODE **yUpdateDeviceList**( string& **errmsg**) |
| m | YRETCODE **yUpdateDeviceList**( NSError** **errmsg**) |
| pas | function **yUpdateDeviceList**( var **errmsg**: string): integer |
| vb | function **yUpdateDeviceList**( ByRef **errmsg** As String) As YRETCODE |
| cs | YRETCODE **UpdateDeviceList**( ref string **errmsg**) |
| java | int **UpdateDeviceList**( ) |
| py | def **UpdateDeviceList**( **errmsg**=None) |

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

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

**Parameters :**

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

**Returns :**

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

## YAPI.UpdateDeviceList_async() yUpdateDeviceList_async() — YAPI

Triggers a (re)detection of connected Yoctopuce modules.

| | |
|---|---|
| js | function **yUpdateDeviceList_async**( **callback**, **context**) |
| nodejs | function **UpdateDeviceList_async**( **callback**, **context**) |

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

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

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## 3.2. Accelerometer function interface

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

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

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

**Global functions**

**yFindAccelerometer(func)**

Retrieves an accelerometer for a given identifier.

**yFirstAccelerometer()**

Starts the enumeration of accelerometers currently accessible.

**`YAccelerometer` methods**

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

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

**accelerometer→describe()**

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

**accelerometer→get_advertisedValue()**

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

**accelerometer→get_currentRawValue()**

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

**accelerometer→get_currentValue()**

Returns the current value of the acceleration.

**accelerometer→get_errorMessage()**

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

**accelerometer→get_errorType()**

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

**accelerometer→get_friendlyName()**

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

**accelerometer→get_functionDescriptor()**

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

**accelerometer→get_functionId()**

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

**accelerometer→get_hardwareId()**

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

**accelerometer→get_highestValue()**

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

**accelerometer→get_logFrequency()**

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

**accelerometer→get_logicalName()**

Returns the logical name of the accelerometer.

**accelerometer→get_lowestValue()**

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

**accelerometer→get_module()**

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

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

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

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

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

**accelerometer→get_reportFrequency()**

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

**accelerometer→get_resolution()**

Returns the resolution of the measured values.

**accelerometer→get_unit()**

Returns the measuring unit for the acceleration.

**accelerometer→get_userData()**

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

**accelerometer→get_xValue()**

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

**accelerometer→get_yValue()**

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

**accelerometer→get_zValue()**

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

**accelerometer→isOnline()**

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

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

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

**accelerometer→load(msValidity)**

Preloads the accelerometer cache with a specified validity duration.

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

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

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

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

**accelerometer→nextAccelerometer()**

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

**accelerometer→registerTimedReportCallback(callback)**

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

**accelerometer→registerValueCallback(callback)**

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

**accelerometer→set_highestValue(newval)**

Changes the recorded maximal value observed.

**accelerometer→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**accelerometer→set_logicalName(newval)**

Changes the logical name of the accelerometer.

**accelerometer→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**accelerometer→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**accelerometer→set_resolution(newval)**

Changes the resolution of the measured physical values.

**accelerometer→set_userData(data)**

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

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

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

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

Retrieves an accelerometer for a given identifier.

| | |
|---|---|
| js | function **yFindAccelerometer**( **func**) |
| nodejs | function **FindAccelerometer**( **func**) |
| php | function **yFindAccelerometer**( **$func**) |
| cpp | YAccelerometer* **yFindAccelerometer**( const string& **func**) |
| m | YAccelerometer* **yFindAccelerometer**( NSString* **func**) |
| pas | function **yFindAccelerometer**( **func**: string): TYAccelerometer |
| vb | function **yFindAccelerometer**( ByVal **func** As String) As YAccelerometer |
| cs | YAccelerometer **FindAccelerometer**( string **func**) |
| java | YAccelerometer **FindAccelerometer**( String **func**) |
| py | def **FindAccelerometer**( **func**) |

The identifier can be specified using several formats:

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

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

**Parameters :**

**func** a string that uniquely characterizes the accelerometer

**Returns :**

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

## YAccelerometer.FirstAccelerometer() yFirstAccelerometer()yFirstAccelerometer() — YAccelerometer

Starts the enumeration of accelerometers currently accessible.

| | |
|---|---|
| js | function **yFirstAccelerometer**( ) |
| nodejs | function **FirstAccelerometer**( ) |
| php | function **yFirstAccelerometer**( ) |
| cpp | YAccelerometer* **yFirstAccelerometer**( ) |
| m | YAccelerometer* **yFirstAccelerometer**( ) |
| pas | function **yFirstAccelerometer**( ): TYAccelerometer |
| vb | function **yFirstAccelerometer**( ) As YAccelerometer |
| cs | YAccelerometer **FirstAccelerometer**( ) |
| java | YAccelerometer **FirstAccelerometer**( ) |
| py | def **FirstAccelerometer**( ) |

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

**Returns :**

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

## accelerometer→calibrateFromPoints()[accelerometer calibrateFromPoints: ] YAccelerometer

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

```
js      function calibrateFromPoints( rawValues, refValues)
nodejs  function calibrateFromPoints( rawValues, refValues)
php     function calibrateFromPoints( $rawValues, $refValues)
cpp     int calibrateFromPoints( vector<double> rawValues,
                                 vector<double> refValues)
m       -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                   : (NSMutableArray*) refValues
pas     function calibrateFromPoints( rawValues: TDoubleArray,
                                      refValues: TDoubleArray): LongInt
vb      procedure calibrateFromPoints( )
cs      int calibrateFromPoints( List<double> rawValues,
                                 List<double> refValues)
java    int calibrateFromPoints( ArrayList<Double> rawValues,
                                 ArrayList<Double> refValues)
py      def calibrateFromPoints( rawValues, refValues)
cmd     YAccelerometer target calibrateFromPoints rawValues refValues
```

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

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→describe()[accelerometer describe] YAccelerometer

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

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

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

**Returns :**

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

## accelerometer→get_advertisedValue() accelerometer→advertisedValue()[accelerometer advertisedValue] YAccelerometer

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YAccelerometer **target** **get_advertisedValue** |

**Returns :**

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

## accelerometer→get_currentRawValue() accelerometer→currentRawValue()[accelerometer currentRawValue] YAccelerometer

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

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YAccelerometer **target** **get_currentRawValue** |

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

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

## accelerometer→get_currentValue() accelerometer→currentValue()[accelerometer currentValue] YAccelerometer

Returns the current value of the acceleration.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YAccelerometer **target** **get_currentValue** |

**Returns :**

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

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

## accelerometer→get_errorMessage() accelerometer→errorMessage()[accelerometer errorMessage] YAccelerometer

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

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

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

## accelerometer→get_errorType() accelerometer→errorType() YAccelerometer

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

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

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

## accelerometer→get_friendlyName() accelerometer→friendlyName()[accelerometer friendlyName] YAccelerometer

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

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

**Returns :**

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

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

## accelerometer→get_functionDescriptor() accelerometer→functionDescriptor()[accelerometer functionDescriptor] YAccelerometer

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

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

**Returns :**

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

## accelerometer→get_functionId() accelerometer→functionId()[accelerometer functionId] YAccelerometer

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

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

## accelerometer→get_hardwareId() accelerometer→hardwareId()[accelerometer hardwareId] YAccelerometer

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

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

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

**Returns :**

a string that uniquely identifies the accelerometer (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## accelerometer→get_highestValue() accelerometer→highestValue()[accelerometer highestValue] YAccelerometer

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

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YAccelerometer **target** **get_highestValue** |

**Returns :**

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

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

## accelerometer→get_logFrequency() accelerometer→logFrequency()[accelerometer logFrequency] YAccelerometer

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

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YAccelerometer **target** **get_logFrequency** |

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

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

## accelerometer→get_logicalName() accelerometer→logicalName()[accelerometer logicalName] YAccelerometer

Returns the logical name of the accelerometer.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YAccelerometer **target** **get_logicalName** |

**Returns :**

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

## accelerometer→get_lowestValue() accelerometer→lowestValue()[accelerometer lowestValue] YAccelerometer

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

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YAccelerometer **target** **get_lowestValue** |

**Returns :**

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

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

## accelerometer→get_module() accelerometer→module()[accelerometer module] YAccelerometer

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

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

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

**Returns :**
an instance of `YModule`

## accelerometer→get_module_async() accelerometer→module_async() YAccelerometer

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

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## accelerometer→get_recordedData() accelerometer→recordedData()[accelerometer recordedData: ] YAccelerometer

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

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YAccelerometer **target** **get_recordedData** **startTime** **endTime** |

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

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

**Parameters :**

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

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

**Returns :**

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

## accelerometer→get_reportFrequency() accelerometer→reportFrequency()[accelerometer reportFrequency] YAccelerometer

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

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YAccelerometer **target** **get_reportFrequency** |

**Returns :**

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

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

## accelerometer→get_resolution() accelerometer→resolution()[accelerometer resolution] YAccelerometer

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YAccelerometer **target** **get_resolution** |

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

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

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

## accelerometer→get_unit() accelerometer→unit()[accelerometer unit] YAccelerometer

Returns the measuring unit for the acceleration.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YAccelerometer **target** **get_unit** |

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

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

## accelerometer→get_userData() accelerometer→userData()[accelerometer userData] YAccelerometer

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

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

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

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

## accelerometer→get_xValue() accelerometer→xValue()[accelerometer xValue] YAccelerometer

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

| | |
|---|---|
| js | function **get_xValue**( ) |
| nodejs | function **get_xValue**( ) |
| php | function **get_xValue**( ) |
| cpp | double **get_xValue**( ) |
| m | -(double) xValue |
| pas | function **get_xValue**( ): double |
| vb | function **get_xValue**( ) As Double |
| cs | double **get_xValue**( ) |
| java | double **get_xValue**( ) |
| py | def **get_xValue**( ) |
| cmd | YAccelerometer **target** **get_xValue** |

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

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

## accelerometer→get_yValue() accelerometer→yValue()[accelerometer yValue] YAccelerometer

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

| | |
|---|---|
| js | function **get_yValue**( ) |
| nodejs | function **get_yValue**( ) |
| php | function **get_yValue**( ) |
| cpp | double **get_yValue**( ) |
| m | -(double) yValue |
| pas | function **get_yValue**( ): double |
| vb | function **get_yValue**( ) As Double |
| cs | double **get_yValue**( ) |
| java | double **get_yValue**( ) |
| py | def **get_yValue**( ) |
| cmd | YAccelerometer **target** **get_yValue** |

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

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

## accelerometer→get_zValue() accelerometer→zValue()[accelerometer zValue] YAccelerometer

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

| | |
|---|---|
| `js` | function **get_zValue**( ) |
| `nodejs` | function **get_zValue**( ) |
| `php` | function **get_zValue**( ) |
| `cpp` | double **get_zValue**( ) |
| `m` | -(double) zValue |
| `pas` | function **get_zValue**( ): double |
| `vb` | function **get_zValue**( ) As Double |
| `cs` | double **get_zValue**( ) |
| `java` | double **get_zValue**( ) |
| `py` | def **get_zValue**( ) |
| `cmd` | YAccelerometer **target** **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.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

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

**Returns :**

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

## accelerometer→isOnline_async() YAccelerometer

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

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

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

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## accelerometer→load()[accelerometer load: ] YAccelerometer

Preloads the accelerometer cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

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

**Parameters :**

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

**Returns :**

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

## accelerometer→loadCalibrationPoints() [accelerometer loadCalibrationPoints: ] YAccelerometer

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

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**,<br>vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues**<br>: (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray,<br>var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**,<br>List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**,<br>ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YAccelerometer **target loadCalibrationPoints rawValues refValues** |

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→load_async() YAccelerometer

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

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

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

**Returns :**

nothing : the result is provided to the callback.

## accelerometer→nextAccelerometer()[accelerometer nextAccelerometer] YAccelerometer

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

| | |
|---|---|
| js | function **nextAccelerometer**( ) |
| nodejs | function **nextAccelerometer**( ) |
| php | function **nextAccelerometer**( ) |
| cpp | YAccelerometer * **nextAccelerometer**( ) |
| m | -(YAccelerometer*) **nextAccelerometer** |
| pas | function **nextAccelerometer**( ): TYAccelerometer |
| vb | function **nextAccelerometer**( ) As YAccelerometer |
| cs | YAccelerometer **nextAccelerometer**( ) |
| java | YAccelerometer **nextAccelerometer**( ) |
| py | def **nextAccelerometer**( ) |

**Returns :**

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

## accelerometer→registerTimedReportCallback() [accelerometer registerTimedReportCallback: ] YAccelerometer

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

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YAccelerometerTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YAccelerometerTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYAccelerometerTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

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

**Parameters :**

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

## accelerometer→registerValueCallback() [accelerometer registerValueCallback: ] YAccelerometer

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YAccelerometerValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YAccelerometerValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYAccelerometerValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## accelerometer→set_highestValue() accelerometer→setHighestValue()[accelerometer setHighestValue: ] YAccelerometer

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YAccelerometer **target** **set_highestValue** **newval** |

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→set_logFrequency() accelerometer→setLogFrequency()[accelerometer setLogFrequency: ] YAccelerometer

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YAccelerometer **target** **set_logFrequency** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→set_logicalName() accelerometer→setLogicalName()[accelerometer setLogicalName: ] YAccelerometer

Changes the logical name of the accelerometer.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YAccelerometer **target** **set_logicalName** **newval** |

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

**Parameters :**

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

**Returns :**

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

## accelerometer→set_lowestValue() accelerometer→setLowestValue()[accelerometer setLowestValue: ] YAccelerometer

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YAccelerometer **target** **set_lowestValue** **newval** |

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→set_reportFrequency() accelerometer→setReportFrequency()[accelerometer setReportFrequency: ] YAccelerometer

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YAccelerometer **target** **set_reportFrequency** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→set_resolution() accelerometer→setResolution()[accelerometer setResolution: ] YAccelerometer

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YAccelerometer **target set_resolution newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## accelerometer→set_userData() accelerometer→setUserData()[accelerometer setUserData: ] YAccelerometer

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

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

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

**Parameters :**

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

## accelerometer→wait_async() YAccelerometer

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

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

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

**Parameters :**

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

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

**Returns :**

nothing.

## 3.3. AnButton function interface

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

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

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

**Global functions**

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

Retrieves an analog input for a given identifier.

**yFirstAnButton**()

Starts the enumeration of analog inputs currently accessible.

**`YAnButton` methods**

**anbutton→describe**()

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

**anbutton→get_advertisedValue**()

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

**anbutton→get_analogCalibration**()

Tells if a calibration process is currently ongoing.

**anbutton→get_calibratedValue**()

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

**anbutton→get_calibrationMax**()

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

**anbutton→get_calibrationMin**()

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

**anbutton→get_errorMessage**()

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

**anbutton→get_errorType**()

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

**anbutton→get_friendlyName**()

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

**anbutton→get_functionDescriptor**()

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

**anbutton→get_functionId()**

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

**anbutton→get_hardwareId()**

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

**anbutton→get_isPressed()**

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

**anbutton→get_lastTimePressed()**

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

**anbutton→get_lastTimeReleased()**

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

**anbutton→get_logicalName()**

Returns the logical name of the analog input.

**anbutton→get_module()**

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

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

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

**anbutton→get_pulseCounter()**

Returns the pulse counter value

**anbutton→get_pulseTimer()**

Returns the timer of the pulses counter (ms)

**anbutton→get_rawValue()**

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

**anbutton→get_sensitivity()**

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

**anbutton→get_userData()**

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

**anbutton→isOnline()**

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

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

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

**anbutton→load(msValidity)**

Preloads the analog input cache with a specified validity duration.

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

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

**anbutton→nextAnButton()**

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

**anbutton→registerValueCallback(callback)**

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

**anbutton→resetCounter()**

Returns the pulse counter value as well as his timer

**anbutton→set_analogCalibration(newval)**

Starts or stops the calibration process.

**anbutton→set_calibrationMax(newval)**

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

**anbutton→set_calibrationMin(newval)**

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

**anbutton→set_logicalName(newval)**

Changes the logical name of the analog input.

**anbutton→set_sensitivity(newval)**

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

**anbutton→set_userData(data)**

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

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

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

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

Retrieves an analog input for a given identifier.

| | |
|---|---|
| js | function **yFindAnButton**( **func**) |
| nodejs | function **FindAnButton**( **func**) |
| php | function **yFindAnButton**( **$func**) |
| cpp | YAnButton* **yFindAnButton**( const string& **func**) |
| m | YAnButton* **yFindAnButton**( NSString* **func**) |
| pas | function **yFindAnButton**( **func**: string): TYAnButton |
| vb | function **yFindAnButton**( ByVal **func** As String) As YAnButton |
| cs | YAnButton **FindAnButton**( string **func**) |
| java | YAnButton **FindAnButton**( String **func**) |
| py | def **FindAnButton**( **func**) |

The identifier can be specified using several formats:

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

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

**Parameters :**

**func** a string that uniquely characterizes the analog input

**Returns :**

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

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

Starts the enumeration of analog inputs currently accessible.

| | |
|---|---|
| js | function **yFirstAnButton**( ) |
| nodejs | function **FirstAnButton**( ) |
| php | function **yFirstAnButton**( ) |
| cpp | YAnButton* **yFirstAnButton**( ) |
| m | YAnButton* **yFirstAnButton**( ) |
| pas | function **yFirstAnButton**( ): TYAnButton |
| vb | function **yFirstAnButton**( ) As YAnButton |
| cs | YAnButton **FirstAnButton**( ) |
| java | YAnButton **FirstAnButton**( ) |
| py | def **FirstAnButton**( ) |

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

**Returns :**

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

## anbutton→describe()[anbutton describe] YAnButton

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

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

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

**Returns :**

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

## anbutton→get_advertisedValue() anbutton→advertisedValue()[anbutton advertisedValue] YAnButton

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YAnButton **target** **get_advertisedValue** |

**Returns :**

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

## anbutton→get_analogCalibration() anbutton→analogCalibration()[anbutton analogCalibration] YAnButton

Tells if a calibration process is currently ongoing.

| | |
|---|---|
| js | function **get_analogCalibration**( ) |
| nodejs | function **get_analogCalibration**( ) |
| php | function **get_analogCalibration**( ) |
| cpp | Y_ANALOGCALIBRATION_enum **get_analogCalibration**( ) |
| m | -(Y_ANALOGCALIBRATION_enum) analogCalibration |
| pas | function **get_analogCalibration**( ): Integer |
| vb | function **get_analogCalibration**( ) As Integer |
| cs | int **get_analogCalibration**( ) |
| java | int **get_analogCalibration**( ) |
| py | def **get_analogCalibration**( ) |
| cmd | YAnButton **target** **get_analogCalibration** |

**Returns :**

either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

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

## anbutton→get_calibratedValue() anbutton→calibratedValue()[anbutton calibratedValue] YAnButton

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

| | |
|---|---|
| js | function **get_calibratedValue**( ) |
| nodejs | function **get_calibratedValue**( ) |
| php | function **get_calibratedValue**( ) |
| cpp | int **get_calibratedValue**( ) |
| m | -(int) calibratedValue |
| pas | function **get_calibratedValue**( ): LongInt |
| vb | function **get_calibratedValue**( ) As Integer |
| cs | int **get_calibratedValue**( ) |
| java | int **get_calibratedValue**( ) |
| py | def **get_calibratedValue**( ) |
| cmd | YAnButton **target** **get_calibratedValue** |

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

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

## anbutton→get_calibrationMax() anbutton→calibrationMax()[anbutton calibrationMax] YAnButton

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

| | |
|---|---|
| js | function **get_calibrationMax**( ) |
| nodejs | function **get_calibrationMax**( ) |
| php | function **get_calibrationMax**( ) |
| cpp | int **get_calibrationMax**( ) |
| m | -(int) calibrationMax |
| pas | function **get_calibrationMax**( ): LongInt |
| vb | function **get_calibrationMax**( ) As Integer |
| cs | int **get_calibrationMax**( ) |
| java | int **get_calibrationMax**( ) |
| py | def **get_calibrationMax**( ) |
| cmd | YAnButton **target** **get_calibrationMax** |

**Returns :**

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

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

## anbutton→get_calibrationMin() anbutton→calibrationMin()[anbutton calibrationMin] YAnButton

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

| | |
|---|---|
| js | function **get_calibrationMin**( ) |
| nodejs | function **get_calibrationMin**( ) |
| php | function **get_calibrationMin**( ) |
| cpp | int **get_calibrationMin**( ) |
| m | -(int) calibrationMin |
| pas | function **get_calibrationMin**( ): LongInt |
| vb | function **get_calibrationMin**( ) As Integer |
| cs | int **get_calibrationMin**( ) |
| java | int **get_calibrationMin**( ) |
| py | def **get_calibrationMin**( ) |
| cmd | YAnButton **target** **get_calibrationMin** |

**Returns :**

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

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

## anbutton→get_errorMessage() anbutton→errorMessage()[anbutton errorMessage] YAnButton

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

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

**Returns :**

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

## anbutton→get_errorType() anbutton→errorType() YAnButton

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

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

**Returns :**

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

## anbutton→get_friendlyName() anbutton→friendlyName()[anbutton friendlyName] YAnButton

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

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

**Returns :**

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

## anbutton→get_functionDescriptor() anbutton→functionDescriptor()[anbutton functionDescriptor] YAnButton

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

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

**Returns :**

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

## anbutton→get_functionId() anbutton→functionId()[anbutton functionId] YAnButton

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

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

## anbutton→get_hardwareId() anbutton→hardwareId()[anbutton hardwareId] YAnButton

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

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

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

**Returns :**

a string that uniquely identifies the analog input (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## anbutton→get_isPressed() anbutton→isPressed()[anbutton isPressed] YAnButton

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

| | |
|---|---|
| js | function **get_isPressed**( ) |
| nodejs | function **get_isPressed**( ) |
| php | function **get_isPressed**( ) |
| cpp | Y_ISPRESSED_enum **get_isPressed**( ) |
| m | -(Y_ISPRESSED_enum) isPressed |
| pas | function **get_isPressed**( ): Integer |
| vb | function **get_isPressed**( ) As Integer |
| cs | int **get_isPressed**( ) |
| java | int **get_isPressed**( ) |
| py | def **get_isPressed**( ) |
| cmd | YAnButton **target** **get_isPressed** |

**Returns :**

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

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

## anbutton→get_lastTimePressed() anbutton→lastTimePressed()[anbutton lastTimePressed] YAnButton

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

| | |
|---|---|
| js | function **get_lastTimePressed**( ) |
| nodejs | function **get_lastTimePressed**( ) |
| php | function **get_lastTimePressed**( ) |
| cpp | s64 **get_lastTimePressed**( ) |
| m | -(s64) lastTimePressed |
| pas | function **get_lastTimePressed**( ): int64 |
| vb | function **get_lastTimePressed**( ) As Long |
| cs | long **get_lastTimePressed**( ) |
| java | long **get_lastTimePressed**( ) |
| py | def **get_lastTimePressed**( ) |
| cmd | YAnButton **target** **get_lastTimePressed** |

**Returns :**

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

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

## anbutton→get_lastTimeReleased() anbutton→lastTimeReleased()[anbutton lastTimeReleased] YAnButton

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

| | |
|---|---|
| js | function **get_lastTimeReleased**( ) |
| nodejs | function **get_lastTimeReleased**( ) |
| php | function **get_lastTimeReleased**( ) |
| cpp | s64 **get_lastTimeReleased**( ) |
| m | -(s64) lastTimeReleased |
| pas | function **get_lastTimeReleased**( ): int64 |
| vb | function **get_lastTimeReleased**( ) As Long |
| cs | long **get_lastTimeReleased**( ) |
| java | long **get_lastTimeReleased**( ) |
| py | def **get_lastTimeReleased**( ) |
| cmd | YAnButton **target** **get_lastTimeReleased** |

**Returns :**

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

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

## anbutton→get_logicalName() anbutton→logicalName()[anbutton logicalName] YAnButton

Returns the logical name of the analog input.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YAnButton **target** **get_logicalName** |

**Returns :**

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

## anbutton→get_module() anbutton→module()[anbutton module] YAnButton

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

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

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

**Returns :**
an instance of `YModule`

## anbutton→get_module_async() anbutton→module_async() YAnButton

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

`js` function **get_module_async**( **callback**, **context**)
`nodejs` function **get_module_async**( **callback**, **context**)

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## anbutton→get_pulseCounter() anbutton→pulseCounter()[anbutton pulseCounter] YAnButton

Returns the pulse counter value

| | |
|---|---|
| js | function **get_pulseCounter**( ) |
| nodejs | function **get_pulseCounter**( ) |
| php | function **get_pulseCounter**( ) |
| cpp | s64 **get_pulseCounter**( ) |
| m | -(s64) pulseCounter |
| pas | function **get_pulseCounter**( ): int64 |
| vb | function **get_pulseCounter**( ) As Long |
| cs | long **get_pulseCounter**( ) |
| java | long **get_pulseCounter**( ) |
| py | def **get_pulseCounter**( ) |

**Returns :**
an integer corresponding to the pulse counter value

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

## anbutton→get_pulseTimer() anbutton→pulseTimer()[anbutton pulseTimer] YAnButton

Returns the timer of the pulses counter (ms)

| | |
|---|---|
| js | function **get_pulseTimer**( ) |
| nodejs | function **get_pulseTimer**( ) |
| php | function **get_pulseTimer**( ) |
| cpp | s64 **get_pulseTimer**( ) |
| m | -(s64) pulseTimer |
| pas | function **get_pulseTimer**( ): int64 |
| vb | function **get_pulseTimer**( ) As Long |
| cs | long **get_pulseTimer**( ) |
| java | long **get_pulseTimer**( ) |
| py | def **get_pulseTimer**( ) |

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

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

## anbutton→get_rawValue() anbutton→rawValue()[anbutton rawValue] YAnButton

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

| | |
|---|---|
| js | function **get_rawValue**( ) |
| nodejs | function **get_rawValue**( ) |
| php | function **get_rawValue**( ) |
| cpp | int **get_rawValue**( ) |
| m | -(int) rawValue |
| pas | function **get_rawValue**( ): LongInt |
| vb | function **get_rawValue**( ) As Integer |
| cs | int **get_rawValue**( ) |
| java | int **get_rawValue**( ) |
| py | def **get_rawValue**( ) |
| cmd | YAnButton **target** **get_rawValue** |

**Returns :**

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

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

## anbutton→get_sensitivity() anbutton→sensitivity()[anbutton sensitivity] YAnButton

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

| | |
|---|---|
| js | function **get_sensitivity**( ) |
| nodejs | function **get_sensitivity**( ) |
| php | function **get_sensitivity**( ) |
| cpp | int **get_sensitivity**( ) |
| m | -(int) sensitivity |
| pas | function **get_sensitivity**( ): LongInt |
| vb | function **get_sensitivity**( ) As Integer |
| cs | int **get_sensitivity**( ) |
| java | int **get_sensitivity**( ) |
| py | def **get_sensitivity**( ) |
| cmd | YAnButton **target** **get_sensitivity** |

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

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

## anbutton→get_userData() anbutton→userData()[anbutton userData] YAnButton

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

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

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

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

## anbutton→isOnline()[anbutton isOnline] YAnButton

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

| | |
|---|---|
| `js` | function **isOnline**( ) |
| `nodejs` | function **isOnline**( ) |
| `php` | function **isOnline**( ) |
| `cpp` | bool **isOnline**( ) |
| `m` | -(BOOL) **isOnline** |
| `pas` | function **isOnline**( ): boolean |
| `vb` | function **isOnline**( ) As Boolean |
| `cs` | bool **isOnline**( ) |
| `java` | boolean **isOnline**( ) |
| `py` | def **isOnline**( ) |

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

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

## anbutton→isOnline_async() YAnButton

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

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

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

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## anbutton→load()[anbutton load: ] YAnButton

Preloads the analog input cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

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

**Parameters :**

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

**Returns :**

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

## anbutton→load_async() YAnButton

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

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

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

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

**Returns :**

nothing : the result is provided to the callback.

## anbutton→nextAnButton()[anbutton nextAnButton] YAnButton

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

| | |
|---|---|
| js | function **nextAnButton**( ) |
| nodejs | function **nextAnButton**( ) |
| php | function **nextAnButton**( ) |
| cpp | YAnButton * **nextAnButton**( ) |
| m | -(YAnButton*) **nextAnButton** |
| pas | function **nextAnButton**( ): TYAnButton |
| vb | function **nextAnButton**( ) As YAnButton |
| cs | YAnButton **nextAnButton**( ) |
| java | YAnButton **nextAnButton**( ) |
| py | def **nextAnButton**( ) |

**Returns :**

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

## anbutton→registerValueCallback()[anbutton registerValueCallback: ] YAnButton

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YAnButtonValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YAnButtonValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYAnButtonValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## anbutton→resetCounter()[anbutton resetCounter] YAnButton

Returns the pulse counter value as well as his timer

| | |
|---|---|
| js | function **resetCounter**( ) |
| nodejs | function **resetCounter**( ) |
| php | function **resetCounter**( ) |
| cpp | int **resetCounter**( ) |
| m | -(int) **resetCounter** |
| pas | function **resetCounter**( ): LongInt |
| vb | function **resetCounter**( ) As Integer |
| cs | int **resetCounter**( ) |
| java | int **resetCounter**( ) |
| py | def **resetCounter**( ) |

**Returns :**
`YAPI_SUCCESS` if the call succeeds.

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

## anbutton→set_analogCalibration() anbutton→setAnalogCalibration()[anbutton setAnalogCalibration: ] YAnButton

Starts or stops the calibration process.

| | |
|---|---|
| js | function **set_analogCalibration**( **newval**) |
| nodejs | function **set_analogCalibration**( **newval**) |
| php | function **set_analogCalibration**( **$newval**) |
| cpp | int **set_analogCalibration**( Y_ANALOGCALIBRATION_enum **newval**) |
| m | -(int) setAnalogCalibration : (Y_ANALOGCALIBRATION_enum) **newval** |
| pas | function **set_analogCalibration**( **newval**: Integer): integer |
| vb | function **set_analogCalibration**( ByVal **newval** As Integer) As Integer |
| cs | int **set_analogCalibration**( int **newval**) |
| java | int **set_analogCalibration**( int **newval**) |
| py | def **set_analogCalibration**( **newval**) |
| cmd | YAnButton **target** **set_analogCalibration** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## anbutton→set_calibrationMax() anbutton→setCalibrationMax()[anbutton setCalibrationMax: ] YAnButton

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

| | |
|---|---|
| js | function **set_calibrationMax**( **newval**) |
| nodejs | function **set_calibrationMax**( **newval**) |
| php | function **set_calibrationMax**( **$newval**) |
| cpp | int **set_calibrationMax**( int **newval**) |
| m | -(int) setCalibrationMax : (int) **newval** |
| pas | function **set_calibrationMax**( **newval**: LongInt): integer |
| vb | function **set_calibrationMax**( ByVal **newval** As Integer) As Integer |
| cs | int **set_calibrationMax**( int **newval**) |
| java | int **set_calibrationMax**( int **newval**) |
| py | def **set_calibrationMax**( **newval**) |
| cmd | YAnButton **target set_calibrationMax newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

### anbutton→set_calibrationMin() anbutton→setCalibrationMin()[anbutton setCalibrationMin: ] — YAnButton

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

| | |
|---|---|
| js | function **set_calibrationMin**( **newval**) |
| nodejs | function **set_calibrationMin**( **newval**) |
| php | function **set_calibrationMin**( **$newval**) |
| cpp | int **set_calibrationMin**( int **newval**) |
| m | -(int) setCalibrationMin : (int) **newval** |
| pas | function **set_calibrationMin**( **newval**: LongInt): integer |
| vb | function **set_calibrationMin**( ByVal **newval** As Integer) As Integer |
| cs | int **set_calibrationMin**( int **newval**) |
| java | int **set_calibrationMin**( int **newval**) |
| py | def **set_calibrationMin**( **newval**) |
| cmd | YAnButton **target** **set_calibrationMin** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## anbutton→set_logicalName() anbutton→setLogicalName()[anbutton setLogicalName: ] YAnButton

Changes the logical name of the analog input.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YAnButton **target** **set_logicalName** **newval** |

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

**Parameters :**

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

**Returns :**

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

## anbutton→set_sensitivity() anbutton→setSensitivity()[anbutton setSensitivity: ] YAnButton

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

| | |
|---|---|
| js | function **set_sensitivity**( **newval**) |
| nodejs | function **set_sensitivity**( **newval**) |
| php | function **set_sensitivity**( **$newval**) |
| cpp | int **set_sensitivity**( int **newval**) |
| m | -(int) setSensitivity : (int) **newval** |
| pas | function **set_sensitivity**( **newval**: LongInt): integer |
| vb | function **set_sensitivity**( ByVal **newval** As Integer) As Integer |
| cs | int **set_sensitivity**( int **newval**) |
| java | int **set_sensitivity**( int **newval**) |
| py | def **set_sensitivity**( **newval**) |
| cmd | YAnButton **target** **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() anbutton→setUserData()[anbutton setUserData: ] YAnButton

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

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

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

**Parameters :**

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

## anbutton→wait_async() YAnButton

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

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

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

**Parameters :**

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

**Returns :**

nothing.

## 3.4. CarbonDioxide function interface

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

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

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

**Global functions**

**yFindCarbonDioxide(func)**

Retrieves a CO2 sensor for a given identifier.

**yFirstCarbonDioxide()**

Starts the enumeration of CO2 sensors currently accessible.

**`YCarbonDioxide` methods**

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

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

**carbondioxide→describe()**

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

**carbondioxide→get_advertisedValue()**

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

**carbondioxide→get_currentRawValue()**

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

**carbondioxide→get_currentValue()**

Returns the current value of the CO2 concentration.

**carbondioxide→get_errorMessage()**

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

**carbondioxide→get_errorType()**

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

**carbondioxide→get_friendlyName()**

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

**carbondioxide→get_functionDescriptor()**

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

**carbondioxide→get_functionId()**

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

**carbondioxide→get_hardwareId()**

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

**carbondioxide→get_highestValue()**

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

**carbondioxide→get_logFrequency()**

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

**carbondioxide→get_logicalName()**

Returns the logical name of the CO2 sensor.

**carbondioxide→get_lowestValue()**

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

**carbondioxide→get_module()**

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

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

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

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

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

**carbondioxide→get_reportFrequency()**

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

**carbondioxide→get_resolution()**

Returns the resolution of the measured values.

**carbondioxide→get_unit()**

Returns the measuring unit for the CO2 concentration.

**carbondioxide→get_userData()**

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

**carbondioxide→isOnline()**

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

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

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

**carbondioxide→load(msValidity)**

Preloads the CO2 sensor cache with a specified validity duration.

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

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

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

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

**carbondioxide→nextCarbonDioxide()**

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

**carbondioxide→registerTimedReportCallback(callback)**

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

**carbondioxide→registerValueCallback(callback)**

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

**carbondioxide→set_highestValue(newval)**

Changes the recorded maximal value observed.

**carbondioxide→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**carbondioxide→set_logicalName(newval)**

Changes the logical name of the CO2 sensor.

**carbondioxide→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**carbondioxide→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**carbondioxide→set_resolution(newval)**

Changes the resolution of the measured physical values.

**carbondioxide→set_userData(data)**

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

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

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

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

Retrieves a CO2 sensor for a given identifier.

| | |
|---|---|
| js | function **yFindCarbonDioxide**( **func**) |
| nodejs | function **FindCarbonDioxide**( **func**) |
| php | function **yFindCarbonDioxide**( **$func**) |
| cpp | YCarbonDioxide* **yFindCarbonDioxide**( const string& **func**) |
| m | YCarbonDioxide* **yFindCarbonDioxide**( NSString* **func**) |
| pas | function **yFindCarbonDioxide**( **func**: string): TYCarbonDioxide |
| vb | function **yFindCarbonDioxide**( ByVal **func** As String) As YCarbonDioxide |
| cs | YCarbonDioxide **FindCarbonDioxide**( string **func**) |
| java | YCarbonDioxide **FindCarbonDioxide**( String **func**) |
| py | def **FindCarbonDioxide**( **func**) |

The identifier can be specified using several formats:

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

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

**Parameters :**

**func** a string that uniquely characterizes the CO2 sensor

**Returns :**

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

## YCarbonDioxide.FirstCarbonDioxide() yFirstCarbonDioxide()yFirstCarbonDioxide()

YCarbonDioxide

Starts the enumeration of CO2 sensors currently accessible.

| | |
|---|---|
| js | function **yFirstCarbonDioxide**( ) |
| nodejs | function **FirstCarbonDioxide**( ) |
| php | function **yFirstCarbonDioxide**( ) |
| cpp | YCarbonDioxide* **yFirstCarbonDioxide**( ) |
| m | YCarbonDioxide* **yFirstCarbonDioxide**( ) |
| pas | function **yFirstCarbonDioxide**( ): TYCarbonDioxide |
| vb | function **yFirstCarbonDioxide**( ) As YCarbonDioxide |
| cs | YCarbonDioxide **FirstCarbonDioxide**( ) |
| java | YCarbonDioxide **FirstCarbonDioxide**( ) |
| py | def **FirstCarbonDioxide**( ) |

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

**Returns :**

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

## carbondioxide→calibrateFromPoints()[carbondioxide calibrateFromPoints: ] YCarbonDioxide

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

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YCarbonDioxide **target calibrateFromPoints rawValues refValues** |

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

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→describe()[carbondioxide describe] YCarbonDioxide

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

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

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

**Returns :**

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

## carbondioxide→get_advertisedValue() carbondioxide→advertisedValue()[carbondioxide advertisedValue] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YCarbonDioxide **target** **get_advertisedValue** |

**Returns :**

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

## carbondioxide→get_currentRawValue() carbondioxide→currentRawValue()[carbondioxide currentRawValue] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YCarbonDioxide **target** **get_currentRawValue** |

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

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

## carbondioxide→get_currentValue() carbondioxide→currentValue()[carbondioxide currentValue]

## YCarbonDioxide

Returns the current value of the CO2 concentration.

| | |
|---|---|
| `js` | function **get_currentValue**( ) |
| `nodejs` | function **get_currentValue**( ) |
| `php` | function **get_currentValue**( ) |
| `cpp` | double **get_currentValue**( ) |
| `m` | -(double) currentValue |
| `pas` | function **get_currentValue**( ): double |
| `vb` | function **get_currentValue**( ) As Double |
| `cs` | double **get_currentValue**( ) |
| `java` | double **get_currentValue**( ) |
| `py` | def **get_currentValue**( ) |
| `cmd` | YCarbonDioxide **target** **get_currentValue** |

**Returns :**

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

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

## carbondioxide→get_errorMessage() carbondioxide→errorMessage()[carbondioxide errorMessage] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

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

**Returns :**

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

## carbondioxide→get_errorType() carbondioxide→errorType() YCarbonDioxide

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

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

**Returns :**

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

## carbondioxide→get_friendlyName() carbondioxide→friendlyName()[carbondioxide friendlyName] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

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

**Returns :**

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

## carbondioxide→get_functionDescriptor() carbondioxide→functionDescriptor()[carbondioxide functionDescriptor] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

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

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

## carbondioxide→get_functionId() carbondioxide→functionId()[carbondioxide functionId] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

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

## carbondioxide→get_hardwareId() carbondioxide→hardwareId()[carbondioxide hardwareId] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the CO2 sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the CO2 sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## carbondioxide→get_highestValue() carbondioxide→highestValue()[carbondioxide highestValue] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YCarbonDioxide **target** **get_highestValue** |

**Returns :**

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

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

## carbondioxide→get_logFrequency() carbondioxide→logFrequency()[carbondioxide logFrequency] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YCarbonDioxide **target** **get_logFrequency** |

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

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

## carbondioxide→get_logicalName() carbondioxide→logicalName()[carbondioxide logicalName] YCarbonDioxide

Returns the logical name of the CO2 sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YCarbonDioxide **target** **get_logicalName** |

**Returns :**

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

## carbondioxide→get_lowestValue() carbondioxide→lowestValue()[carbondioxide lowestValue] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YCarbonDioxide **target** **get_lowestValue** |

**Returns :**

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

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

## carbondioxide→get_module() carbondioxide→module()[carbondioxide module] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

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

**Returns :**
an instance of `YModule`

## carbondioxide→get_module_async() carbondioxide→module_async() YCarbonDioxide

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

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## carbondioxide→get_recordedData() carbondioxide→recordedData()[carbondioxide recordedData: ] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YCarbonDioxide **target** **get_recordedData** **startTime** **endTime** |

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

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

**Parameters :**

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

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

**Returns :**

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

## carbondioxide→get_reportFrequency() carbondioxide→reportFrequency()[carbondioxide reportFrequency] YCarbonDioxide

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

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YCarbonDioxide **target** **get_reportFrequency** |

**Returns :**

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

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

## carbondioxide→get_resolution() carbondioxide→resolution()[carbondioxide resolution] YCarbonDioxide

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YCarbonDioxide **target** **get_resolution** |

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

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

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

## carbondioxide→get_unit() carbondioxide→unit()[carbondioxide unit] YCarbonDioxide

Returns the measuring unit for the CO2 concentration.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YCarbonDioxide **target** **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 userData]

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

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

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

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

## carbondioxide→isOnline()[carbondioxide isOnline] YCarbonDioxide

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

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

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

**Returns :**
`true` if the CO2 sensor can be reached, and `false` otherwise

## carbondioxide→isOnline_async() YCarbonDioxide

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

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

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

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## carbondioxide→load()[carbondioxide load: ] YCarbonDioxide

Preloads the CO2 sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

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

**Parameters :**

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

**Returns :**

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

## carbondioxide→loadCalibrationPoints() [carbondioxide loadCalibrationPoints: ] YCarbonDioxide

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

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YCarbonDioxide **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→load_async() YCarbonDioxide

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

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

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

**Returns :**

nothing : the result is provided to the callback.

## carbondioxide→nextCarbonDioxide()[carbondioxide nextCarbonDioxide] YCarbonDioxide

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

| | |
|---|---|
| js | function **nextCarbonDioxide**( ) |
| nodejs | function **nextCarbonDioxide**( ) |
| php | function **nextCarbonDioxide**( ) |
| cpp | YCarbonDioxide * **nextCarbonDioxide**( ) |
| m | -(YCarbonDioxide*) **nextCarbonDioxide** |
| pas | function **nextCarbonDioxide**( ): TYCarbonDioxide |
| vb | function **nextCarbonDioxide**( ) As YCarbonDioxide |
| cs | YCarbonDioxide **nextCarbonDioxide**( ) |
| java | YCarbonDioxide **nextCarbonDioxide**( ) |
| py | def **nextCarbonDioxide**( ) |

**Returns :**

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

## carbondioxide→registerTimedReportCallback() [carbondioxide registerTimedReportCallback: ] YCarbonDioxide

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

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YCarbonDioxideTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YCarbonDioxideTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYCarbonDioxideTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

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

**Parameters :**

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

### carbondioxide→registerValueCallback() [carbondioxide registerValueCallback: ] YCarbonDioxide

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YCarbonDioxideValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YCarbonDioxideValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYCarbonDioxideValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## carbondioxide→set_highestValue() carbondioxide→setHighestValue()[carbondioxide setHighestValue: ] YCarbonDioxide

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YCarbonDioxide **target** **set_highestValue** **newval** |

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→set_logFrequency() carbondioxide→setLogFrequency()[carbondioxide setLogFrequency: ] YCarbonDioxide

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YCarbonDioxide **target** **set_logFrequency** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→set_logicalName() carbondioxide→setLogicalName()[carbondioxide setLogicalName: ] YCarbonDioxide

Changes the logical name of the CO2 sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YCarbonDioxide **target** **set_logicalName** **newval** |

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

**Parameters :**

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

**Returns :**

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

## carbondioxide→set_lowestValue() carbondioxide→setLowestValue()[carbondioxide setLowestValue: ] YCarbonDioxide

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YCarbonDioxide **target set_lowestValue newval** |

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→set_reportFrequency() carbondioxide→setReportFrequency()[carbondioxide setReportFrequency: ]

**YCarbonDioxide**

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YCarbonDioxide **target** **set_reportFrequency** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→set_resolution() carbondioxide→setResolution()[carbondioxide setResolution: ] YCarbonDioxide

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YCarbonDioxide **target** **set_resolution** **newval** |

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## carbondioxide→set_userData() carbondioxide→setUserData()[carbondioxide setUserData: ] YCarbonDioxide

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

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

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

**Parameters :**

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

### carbondioxide→wait_async() YCarbonDioxide

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

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

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

**Parameters :**

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

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

**Returns :**

nothing.

# 3.5. ColorLed function interface

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

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

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

**Global functions**

**yFindColorLed(func)**

Retrieves an RGB led for a given identifier.

**yFirstColorLed()**

Starts the enumeration of RGB leds currently accessible.

**`YColorLed` methods**

**colorled→describe()**

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

**colorled→get_advertisedValue()**

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

**colorled→get_errorMessage()**

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

**colorled→get_errorType()**

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

**colorled→get_friendlyName()**

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

**colorled→get_functionDescriptor()**

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

**colorled→get_functionId()**

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

**colorled→get_hardwareId()**

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

**colorled→get_hslColor()**

Returns the current HSL color of the led.

**colorled→get_logicalName()**

Returns the logical name of the RGB led.

**colorled→get_module()**

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

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

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

**colorled→get_rgbColor()**

Returns the current RGB color of the led.

**colorled→get_rgbColorAtPowerOn()**

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

**colorled→get_userData()**

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

**colorled→hslMove(hsl_target, ms_duration)**

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

**colorled→isOnline()**

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

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

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

**colorled→load(msValidity)**

Preloads the RGB led cache with a specified validity duration.

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

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

**colorled→nextColorLed()**

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

**colorled→registerValueCallback(callback)**

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

**colorled→rgbMove(rgb_target, ms_duration)**

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

**colorled→set_hslColor(newval)**

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

**colorled→set_logicalName(newval)**

Changes the logical name of the RGB led.

**colorled→set_rgbColor(newval)**

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

**colorled→set_rgbColorAtPowerOn(newval)**

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

**colorled→set_userData(data)**

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

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

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

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

Retrieves an RGB led for a given identifier.

| | |
|---|---|
| js | function **yFindColorLed**( **func**) |
| nodejs | function **FindColorLed**( **func**) |
| php | function **yFindColorLed**( **$func**) |
| cpp | YColorLed* **yFindColorLed**( const string& **func**) |
| m | YColorLed* **yFindColorLed**( NSString* **func**) |
| pas | function **yFindColorLed**( **func**: string): TYColorLed |
| vb | function **yFindColorLed**( ByVal **func** As String) As YColorLed |
| cs | YColorLed **FindColorLed**( string **func**) |
| java | YColorLed **FindColorLed**( String **func**) |
| py | def **FindColorLed**( **func**) |

The identifier can be specified using several formats:

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

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

**Parameters :**

**func** a string that uniquely characterizes the RGB led

**Returns :**

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

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

Starts the enumeration of RGB leds currently accessible.

| | |
|---|---|
| js | function **yFirstColorLed**( ) |
| nodejs | function **FirstColorLed**( ) |
| php | function **yFirstColorLed**( ) |
| cpp | YColorLed* **yFirstColorLed**( ) |
| m | YColorLed* **yFirstColorLed**( ) |
| pas | function **yFirstColorLed**( ): TYColorLed |
| vb | function **yFirstColorLed**( ) As YColorLed |
| cs | YColorLed **FirstColorLed**( ) |
| java | YColorLed **FirstColorLed**( ) |
| py | def **FirstColorLed**( ) |

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

**Returns :**

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

## colorled→describe()[colorled describe] YColorLed

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

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

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

**Returns :**
a string that describes the RGB led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## colorled→get_advertisedValue() colorled→advertisedValue()[colorled advertisedValue] YColorLed

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YColorLed **target** **get_advertisedValue** |

**Returns :**

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

## colorled→get_errorMessage() colorled→errorMessage()[colorled errorMessage] YColorLed

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

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

**Returns :**

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

## colorled→get_errorType() colorled→errorType() YColorLed

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

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

**Returns :**

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

## colorled→get_friendlyName() colorled→friendlyName()[colorled friendlyName] YColorLed

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

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

**Returns :**

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

## colorled→get_functionDescriptor() colorled→functionDescriptor()[colorled functionDescriptor] YColorLed

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

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

**Returns :**

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

## colorled→get_functionId() colorled→functionId()[colorled functionId] YColorLed

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

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

## colorled→get_hardwareId() colorled→hardwareId()[colorled hardwareId] YColorLed

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

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

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

**Returns :**
a string that uniquely identifies the RGB led (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## colorled→get_hslColor() colorled→hslColor()[colorled hslColor] YColorLed

Returns the current HSL color of the led.

| | |
|---|---|
| js | function **get_hslColor**( ) |
| nodejs | function **get_hslColor**( ) |
| php | function **get_hslColor**( ) |
| cpp | int **get_hslColor**( ) |
| m | -(int) hslColor |
| pas | function **get_hslColor**( ): LongInt |
| vb | function **get_hslColor**( ) As Integer |
| cs | int **get_hslColor**( ) |
| java | int **get_hslColor**( ) |
| py | def **get_hslColor**( ) |
| cmd | YColorLed **target** **get_hslColor** |

**Returns :**
an integer corresponding to the current HSL color of the led

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

## colorled→get_logicalName() colorled→logicalName()[colorled logicalName] YColorLed

Returns the logical name of the RGB led.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YColorLed **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the RGB led. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## colorled→get_module() colorled→module()[colorled module] YColorLed

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

| | |
|---|---|
| `js` | function **get_module**( ) |
| `nodejs` | function **get_module**( ) |
| `php` | function **get_module**( ) |
| `cpp` | YModule * **get_module**( ) |
| `m` | -(YModule*) module |
| `pas` | function **get_module**( ): TYModule |
| `vb` | function **get_module**( ) As YModule |
| `cs` | YModule **get_module**( ) |
| `java` | YModule **get_module**( ) |
| `py` | def **get_module**( ) |

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

**Returns :**
an instance of `YModule`

## colorled→get_module_async() colorled→module_async() YColorLed

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

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## colorled→get_rgbColor() colorled→rgbColor()[colorled rgbColor] YColorLed

Returns the current RGB color of the led.

| | |
|---|---|
| js | function **get_rgbColor**( ) |
| nodejs | function **get_rgbColor**( ) |
| php | function **get_rgbColor**( ) |
| cpp | int **get_rgbColor**( ) |
| m | -(int) rgbColor |
| pas | function **get_rgbColor**( ): LongInt |
| vb | function **get_rgbColor**( ) As Integer |
| cs | int **get_rgbColor**( ) |
| java | int **get_rgbColor**( ) |
| py | def **get_rgbColor**( ) |
| cmd | YColorLed **target** **get_rgbColor** |

**Returns :**
an integer corresponding to the current RGB color of the led

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

## colorled→get_rgbColorAtPowerOn() colorled→rgbColorAtPowerOn()[colorled rgbColorAtPowerOn] YColorLed

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

| | |
|---|---|
| js | function **get_rgbColorAtPowerOn**( ) |
| nodejs | function **get_rgbColorAtPowerOn**( ) |
| php | function **get_rgbColorAtPowerOn**( ) |
| cpp | int **get_rgbColorAtPowerOn**( ) |
| m | -(int) rgbColorAtPowerOn |
| pas | function **get_rgbColorAtPowerOn**( ): LongInt |
| vb | function **get_rgbColorAtPowerOn**( ) As Integer |
| cs | int **get_rgbColorAtPowerOn**( ) |
| java | int **get_rgbColorAtPowerOn**( ) |
| py | def **get_rgbColorAtPowerOn**( ) |
| cmd | YColorLed **target** **get_rgbColorAtPowerOn** |

**Returns :**

an integer corresponding to the configured color to be displayed when the module is turned on

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

## colorled→get_userData() colorled→userData()[colorled userData] YColorLed

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

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

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

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

## colorled→hslMove()[colorled hslMove: ] YColorLed

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

| | |
|---|---|
| js | function **hslMove**( **hsl_target**, **ms_duration**) |
| nodejs | function **hslMove**( **hsl_target**, **ms_duration**) |
| php | function **hslMove**( **$hsl_target**, **$ms_duration**) |
| cpp | int **hslMove**( int **hsl_target**, int **ms_duration**) |
| m | -(int) **hslMove** : (int) **hsl_target** : (int) **ms_duration** |
| pas | function **hslMove**( **hsl_target**: LongInt, **ms_duration**: LongInt): integer |
| vb | function **hslMove**( ByVal **hsl_target** As Integer,<br>ByVal **ms_duration** As Integer) As Integer |
| cs | int **hslMove**( int **hsl_target**, int **ms_duration**) |
| java | int **hslMove**( int **hsl_target**, int **ms_duration**) |
| py | def **hslMove**( **hsl_target**, **ms_duration**) |
| cmd | YColorLed **target** **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.

| | |
|---|---|
| `js` | function **isOnline**( ) |
| `nodejs` | function **isOnline**( ) |
| `php` | function **isOnline**( ) |
| `cpp` | bool **isOnline**( ) |
| `m` | -(BOOL) **isOnline** |
| `pas` | function **isOnline**( ): boolean |
| `vb` | function **isOnline**( ) As Boolean |
| `cs` | bool **isOnline**( ) |
| `java` | boolean **isOnline**( ) |
| `py` | def **isOnline**( ) |

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

**Returns :**
`true` if the RGB led can be reached, and `false` otherwise

## colorled→isOnline_async() YColorLed

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

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

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

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## colorled→load()[colorled load: ] YColorLed

Preloads the RGB led cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

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

**Parameters :**

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

**Returns :**

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

## colorled→load_async() YColorLed

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

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

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

**Returns :**

nothing : the result is provided to the callback.

## colorled→nextColorLed()[colorled nextColorLed] YColorLed

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

| | |
|---|---|
| js | function **nextColorLed**( ) |
| nodejs | function **nextColorLed**( ) |
| php | function **nextColorLed**( ) |
| cpp | YColorLed * **nextColorLed**( ) |
| m | -(YColorLed*) **nextColorLed** |
| pas | function **nextColorLed**( ): TYColorLed |
| vb | function **nextColorLed**( ) As YColorLed |
| cs | YColorLed **nextColorLed**( ) |
| java | YColorLed **nextColorLed**( ) |
| py | def **nextColorLed**( ) |

**Returns :**

a pointer to a `YColorLed` object, corresponding to an RGB led currently online, or a `null` pointer if there are no more RGB leds to enumerate.

## colorled→registerValueCallback()[colorled registerValueCallback: ] YColorLed

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YColorLedValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YColorLedValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYColorLedValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## colorled→rgbMove()[colorled rgbMove: ] YColorLed

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

| | |
|---|---|
| js | function **rgbMove**( **rgb_target**, **ms_duration**) |
| nodejs | function **rgbMove**( **rgb_target**, **ms_duration**) |
| php | function **rgbMove**( **$rgb_target**, **$ms_duration**) |
| cpp | int **rgbMove**( int **rgb_target**, int **ms_duration**) |
| m | -(int) **rgbMove** : (int) **rgb_target** : (int) **ms_duration** |
| pas | function **rgbMove**( **rgb_target**: LongInt, **ms_duration**: LongInt): integer |
| vb | function **rgbMove**( ByVal **rgb_target** As Integer, ByVal **ms_duration** As Integer) As Integer |
| cs | int **rgbMove**( int **rgb_target**, int **ms_duration**) |
| java | int **rgbMove**( int **rgb_target**, int **ms_duration**) |
| py | def **rgbMove**( **rgb_target**, **ms_duration**) |
| cmd | YColorLed **target** **rgbMove** **rgb_target** **ms_duration** |

**Parameters :**

**rgb_target** desired RGB color at the end of the transition

**ms_duration** duration of the transition, in millisecond

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## colorled→set_hslColor() colorled→setHslColor()[colorled setHslColor: ] YColorLed

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

| | |
|---|---|
| js | function **set_hslColor**( **newval**) |
| nodejs | function **set_hslColor**( **newval**) |
| php | function **set_hslColor**( **$newval**) |
| cpp | int **set_hslColor**( int **newval**) |
| m | -(int) setHslColor : (int) **newval** |
| pas | function **set_hslColor**( **newval**: LongInt): integer |
| vb | function **set_hslColor**( ByVal **newval** As Integer) As Integer |
| cs | int **set_hslColor**( int **newval**) |
| java | int **set_hslColor**( int **newval**) |
| py | def **set_hslColor**( **newval**) |
| cmd | YColorLed **target** **set_hslColor** **newval** |

Encoding is done as follows: 0xHHSSLL.

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## colorled→set_logicalName() colorled→setLogicalName()[colorled setLogicalName: ] — YColorLed

Changes the logical name of the RGB led.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YColorLed **target** **set_logicalName** **newval** |

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

**Parameters :**

**newval** a string corresponding to the logical name of the RGB led.

**Returns :**

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

## colorled→set_rgbColor() colorled→setRgbColor()[colorled setRgbColor: ] YColorLed

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

| | |
|---|---|
| js | function **set_rgbColor**( **newval**) |
| nodejs | function **set_rgbColor**( **newval**) |
| php | function **set_rgbColor**( **$newval**) |
| cpp | int **set_rgbColor**( int **newval**) |
| m | -(int) setRgbColor : (int) **newval** |
| pas | function **set_rgbColor**( **newval**: LongInt): integer |
| vb | function **set_rgbColor**( ByVal **newval** As Integer) As Integer |
| cs | int **set_rgbColor**( int **newval**) |
| java | int **set_rgbColor**( int **newval**) |
| py | def **set_rgbColor**( **newval**) |
| cmd | YColorLed **target set_rgbColor newval** |

Encoding is done as follows: 0xRRGGBB.

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## colorled→set_rgbColorAtPowerOn() colorled→setRgbColorAtPowerOn()[colorled setRgbColorAtPowerOn: ] YColorLed

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

| | |
|---|---|
| js | function **set_rgbColorAtPowerOn**( **newval**) |
| nodejs | function **set_rgbColorAtPowerOn**( **newval**) |
| php | function **set_rgbColorAtPowerOn**( **$newval**) |
| cpp | int **set_rgbColorAtPowerOn**( int **newval**) |
| m | -(int) setRgbColorAtPowerOn : (int) **newval** |
| pas | function **set_rgbColorAtPowerOn**( **newval**: LongInt): integer |
| vb | function **set_rgbColorAtPowerOn**( ByVal **newval** As Integer) As Integer |
| cs | int **set_rgbColorAtPowerOn**( int **newval**) |
| java | int **set_rgbColorAtPowerOn**( int **newval**) |
| py | def **set_rgbColorAtPowerOn**( **newval**) |
| cmd | YColorLed **target** **set_rgbColorAtPowerOn** **newval** |

This color will be displayed as soon as the module is powered on. Remember to call the `saveToFlash()` method of the module if the change should be kept.

**Parameters :**

**newval** an integer corresponding to the color that the led will display by default when the module is turned on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## colorled→set_userData() colorled→setUserData()[colorled setUserData: ] YColorLed

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

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

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

**Parameters :**

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

## colorled→wait_async() YColorLed

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

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

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

**Parameters :**

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

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

**Returns :**

nothing.

## 3.6. Compass function interface

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

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

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

**Global functions**

**yFindCompass(func)**
Retrieves a compass for a given identifier.

**yFirstCompass()**
Starts the enumeration of compasses currently accessible.

**YCompass methods**

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

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

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

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

**compass→get_currentValue()**
Returns the current value of the relative bearing.

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

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

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

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

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

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

**compass→get_highestValue**()

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

**compass→get_logFrequency**()

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

**compass→get_logicalName**()

Returns the logical name of the compass.

**compass→get_lowestValue**()

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

**compass→get_magneticHeading**()

Returns the magnetic heading, regardless of the configured bearing.

**compass→get_module**()

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

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

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

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

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

**compass→get_reportFrequency**()

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

**compass→get_resolution**()

Returns the resolution of the measured values.

**compass→get_unit**()

Returns the measuring unit for the relative bearing.

**compass→get_userData**()

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

**compass→isOnline**()

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

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

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

**compass→load**(**msValidity**)

Preloads the compass cache with a specified validity duration.

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

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

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

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

**compass→nextCompass**()

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

**compass→registerTimedReportCallback**(**callback**)

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

**compass→registerValueCallback**(**callback**)

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

**compass→set_highestValue**(**newval**)

Changes the recorded maximal value observed.

**compass→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**compass→set_logicalName(newval)**

Changes the logical name of the compass.

**compass→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**compass→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**compass→set_resolution(newval)**

Changes the resolution of the measured physical values.

**compass→set_userData(data)**

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

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

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

## YCompass.FindCompass() yFindCompass()yFindCompass()

## YCompass

Retrieves a compass for a given identifier.

| | |
|---|---|
| js | function **yFindCompass**( **func**) |
| nodejs | function **FindCompass**( **func**) |
| php | function **yFindCompass**( **$func**) |
| cpp | YCompass* **yFindCompass**( const string& **func**) |
| m | YCompass* **yFindCompass**( NSString* **func**) |
| pas | function **yFindCompass**( **func**: string): TYCompass |
| vb | function **yFindCompass**( ByVal **func** As String) As YCompass |
| cs | YCompass **FindCompass**( string **func**) |
| java | YCompass **FindCompass**( String **func**) |
| py | def **FindCompass**( **func**) |

The identifier can be specified using several formats:

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

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

**Parameters :**

**func** a string that uniquely characterizes the compass

**Returns :**

a `YCompass` object allowing you to drive the compass.

## YCompass.FirstCompass() yFirstCompass()yFirstCompass() YCompass

Starts the enumeration of compasses currently accessible.

| | |
|---|---|
| js | function **yFirstCompass**( ) |
| nodejs | function **FirstCompass**( ) |
| php | function **yFirstCompass**( ) |
| cpp | YCompass* **yFirstCompass**( ) |
| m | YCompass* **yFirstCompass**( ) |
| pas | function **yFirstCompass**( ): TYCompass |
| vb | function **yFirstCompass**( ) As YCompass |
| cs | YCompass **FirstCompass**( ) |
| java | YCompass **FirstCompass**( ) |
| py | def **FirstCompass**( ) |

Use the method `YCompass.nextCompass()` to iterate on next compasses.

**Returns :**

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

## compass→calibrateFromPoints()[compass calibrateFromPoints: ] YCompass

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

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YCompass **target** **calibrateFromPoints** **rawValues** **refValues** |

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

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## compass→describe()[compass describe] YCompass

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

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

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

**Returns :**

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

## compass→get_advertisedValue() compass→advertisedValue()[compass advertisedValue]

**YCompass**

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YCompass **target** **get_advertisedValue** |

**Returns :**

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

## compass→get_currentRawValue() compass→currentRawValue()[compass currentRawValue] YCompass

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

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YCompass **target** **get_currentRawValue** |

**Returns :**

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

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

## compass→get_currentValue() compass→currentValue()[compass currentValue] YCompass

Returns the current value of the relative bearing.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YCompass **target** **get_currentValue** |

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

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

## compass→get_errorMessage() compass→errorMessage()[compass errorMessage] YCompass

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

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

**Returns :**

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

## compass→get_errorType() compass→errorType()

**YCompass**

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

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

**Returns :**
a number corresponding to the code of the latest error that occured while using the compass object

## compass→get_friendlyName() compass→friendlyName()[compass friendlyName] YCompass

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

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

**Returns :**

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

## compass→get_functionDescriptor() compass→functionDescriptor()[compass functionDescriptor] YCompass

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

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

**Returns :**

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

## compass→get_functionId() compass→functionId()[compass functionId] — YCompass

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

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

## compass→get_hardwareId() compass→hardwareId()[compass hardwareId] YCompass

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

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

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

**Returns :**

a string that uniquely identifies the compass (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## compass→get_highestValue() compass→highestValue()[compass highestValue] YCompass

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

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YCompass **target** **get_highestValue** |

**Returns :**

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

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

## compass→get_logFrequency() compass→logFrequency()[compass logFrequency] YCompass

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

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YCompass **target** **get_logFrequency** |

**Returns :**

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

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

## compass→get_logicalName() compass→logicalName()[compass logicalName] YCompass

Returns the logical name of the compass.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YCompass **target** **get_logicalName** |

**Returns :**

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

## compass→get_lowestValue() compass→lowestValue()[compass lowestValue] YCompass

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

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YCompass **target** **get_lowestValue** |

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

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

## compass→get_magneticHeading() compass→magneticHeading()[compass magneticHeading] YCompass

Returns the magnetic heading, regardless of the configured bearing.

| | |
|---|---|
| js | function **get_magneticHeading**( ) |
| nodejs | function **get_magneticHeading**( ) |
| php | function **get_magneticHeading**( ) |
| cpp | double **get_magneticHeading**( ) |
| m | -(double) magneticHeading |
| pas | function **get_magneticHeading**( ): double |
| vb | function **get_magneticHeading**( ) As Double |
| cs | double **get_magneticHeading**( ) |
| java | double **get_magneticHeading**( ) |
| py | def **get_magneticHeading**( ) |
| cmd | YCompass **target** **get_magneticHeading** |

**Returns :**
a floating point number corresponding to the magnetic heading, regardless of the configured bearing

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

## compass→get_module() compass→module()[compass module] YCompass

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

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

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

**Returns :**
an instance of `YModule`

## compass→get_module_async() compass→module_async() — YCompass

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

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## compass→get_recordedData() compass→recordedData()[compass recordedData: ] YCompass

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

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YCompass **target get_recordedData startTime endTime** |

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

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

**Parameters :**

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

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

**Returns :**

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

## compass→get_reportFrequency() compass→reportFrequency()[compass reportFrequency] YCompass

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

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YCompass **target** **get_reportFrequency** |

**Returns :**

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

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

## compass→get_resolution() compass→resolution()[compass resolution] YCompass

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YCompass **target** **get_resolution** |

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

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

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

## compass→get_unit() compass→unit()[compass unit] YCompass

Returns the measuring unit for the relative bearing.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YCompass **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the relative bearing

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

## compass→get_userData() compass→userData()[compass userData] YCompass

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

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

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

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

## compass→isOnline()[compass isOnline] YCompass

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

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

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

**Returns :**

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

## compass→isOnline_async() YCompass

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

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

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

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

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## compass→load()[compass load: ] YCompass

Preloads the compass cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## compass→loadCalibrationPoints()[compass loadCalibrationPoints: ] YCompass

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YCompass **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→load_async() YCompass

Preloads the compass cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## compass→nextCompass()[compass nextCompass] YCompass

Continues the enumeration of compasses started using `yFirstCompass()`.

| | |
|---|---|
| js | function **nextCompass**( ) |
| nodejs | function **nextCompass**( ) |
| php | function **nextCompass**( ) |
| cpp | YCompass * **nextCompass**( ) |
| m | -(YCompass*) **nextCompass** |
| pas | function **nextCompass**( ): TYCompass |
| vb | function **nextCompass**( ) As YCompass |
| cs | YCompass **nextCompass**( ) |
| java | YCompass **nextCompass**( ) |
| py | def **nextCompass**( ) |

**Returns :**

a pointer to a `YCompass` object, corresponding to a compass currently online, or a `null` pointer if there are no more compasses to enumerate.

## compass→registerTimedReportCallback()[compass registerTimedReportCallback: ] YCompass

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YCompassTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YCompassTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYCompassTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## compass→registerValueCallback()[compass registerValueCallback: ] YCompass

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YCompassValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YCompassValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYCompassValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## compass→set_highestValue() compass→setHighestValue()[compass setHighestValue: ] YCompass

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YCompass **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→set_logFrequency() compass→setLogFrequency()[compass setLogFrequency: ] YCompass

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YCompass **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→set_logicalName() compass→setLogicalName()[compass setLogicalName: ] YCompass

Changes the logical name of the compass.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YCompass **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the compass.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## compass→set_lowestValue() compass→setLowestValue()[compass setLowestValue: ] YCompass

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YCompass **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→set_reportFrequency() compass→setReportFrequency()[compass setReportFrequency: ] YCompass

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YCompass **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→set_resolution() compass→setResolution()[compass setResolution: ] — YCompass

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YCompass **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→set_userData() compass→setUserData()[compass setUserData: ]  YCompass

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## compass→wait_async() YCompass

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

- **callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.7. Current function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_current.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YCurrent = yoctolib.YCurrent; |
| php | require_once('yocto_current.php'); |
| cpp | #include "yocto_current.h" |
| m | #import "yocto_current.h" |
| pas | uses yocto_current; |
| vb | yocto_current.vb |
| cs | yocto_current.cs |
| java | import com.yoctopuce.YoctoAPI.YCurrent; |
| py | from yocto_current import * |

**Global functions**

**yFindCurrent(func)**

Retrieves a current sensor for a given identifier.

**yFirstCurrent()**

Starts the enumeration of current sensors currently accessible.

**`YCurrent` methods**

**current→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**current→describe()**

Returns a short text that describes unambiguously the instance of the current sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**current→get_advertisedValue()**

Returns the current value of the current sensor (no more than 6 characters).

**current→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**current→get_currentValue()**

Returns the current measure for the current.

**current→get_errorMessage()**

Returns the error message of the latest error with the current sensor.

**current→get_errorType()**

Returns the numerical error code of the latest error with the current sensor.

**current→get_friendlyName()**

Returns a global identifier of the current sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**current→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**current→get_functionId()**

Returns the hardware identifier of the current sensor, without reference to the module.

**current→get_hardwareId()**

Returns the unique hardware identifier of the current sensor in the form `SERIAL.FUNCTIONID`.

**current→get_highestValue()**
Returns the maximal value observed for the current.

**current→get_logFrequency()**
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**current→get_logicalName()**
Returns the logical name of the current sensor.

**current→get_lowestValue()**
Returns the minimal value observed for the current.

**current→get_module()**
Gets the `YModule` object for the device on which the function is located.

**current→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**current→get_recordedData(startTime, endTime)**
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**current→get_reportFrequency()**
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**current→get_resolution()**
Returns the resolution of the measured values.

**current→get_unit()**
Returns the measuring unit for the current.

**current→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**current→isOnline()**
Checks if the current sensor is currently reachable, without raising any error.

**current→isOnline_async(callback, context)**
Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

**current→load(msValidity)**
Preloads the current sensor cache with a specified validity duration.

**current→loadCalibrationPoints(rawValues, refValues)**
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**current→load_async(msValidity, callback, context)**
Preloads the current sensor cache with a specified validity duration (asynchronous version).

**current→nextCurrent()**
Continues the enumeration of current sensors started using `yFirstCurrent()`.

**current→registerTimedReportCallback(callback)**
Registers the callback function that is invoked on every periodic timed notification.

**current→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**current→set_highestValue(newval)**
Changes the recorded maximal value observed pour the current.

**current→set_logFrequency(newval)**
Changes the datalogger recording frequency for this function.

**current→set_logicalName(newval)**
Changes the logical name of the current sensor.

**current→set_lowestValue(newval)**

Changes the recorded minimal value observed pour the current.

**current→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**current→set_resolution(newval)**

Changes the resolution of the measured values.

**current→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**current→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YCurrent.FindCurrent() yFindCurrent()yFindCurrent() — YCurrent

Retrieves a current sensor for a given identifier.

| | |
|---|---|
| js | function **yFindCurrent**( **func**) |
| nodejs | function **FindCurrent**( **func**) |
| php | function **yFindCurrent**( **$func**) |
| cpp | YCurrent* **yFindCurrent**( const string& **func**) |
| m | YCurrent* **yFindCurrent**( NSString* **func**) |
| pas | function **yFindCurrent**( **func**: string): TYCurrent |
| vb | function **yFindCurrent**( ByVal **func** As String) As YCurrent |
| cs | YCurrent **FindCurrent**( string **func**) |
| java | YCurrent **FindCurrent**( String **func**) |
| py | def **FindCurrent**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the current sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrent.isOnline()` to test if the current sensor is indeed online at a given time. In case of ambiguity when looking for a current sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the current sensor

**Returns :**

a `YCurrent` object allowing you to drive the current sensor.

## YCurrent.FirstCurrent() yFirstCurrent()yFirstCurrent() YCurrent

Starts the enumeration of current sensors currently accessible.

| | |
|---|---|
| js | function **yFirstCurrent**( ) |
| nodejs | function **FirstCurrent**( ) |
| php | function **yFirstCurrent**( ) |
| cpp | YCurrent* **yFirstCurrent**( ) |
| m | YCurrent* **yFirstCurrent**( ) |
| pas | function **yFirstCurrent**( ): TYCurrent |
| vb | function **yFirstCurrent**( ) As YCurrent |
| cs | YCurrent **FirstCurrent**( ) |
| java | YCurrent **FirstCurrent**( ) |
| py | def **FirstCurrent**( ) |

Use the method `YCurrent.nextCurrent()` to iterate on next current sensors.

**Returns :**

a pointer to a `YCurrent` object, corresponding to the first current sensor currently online, or a `null` pointer if there are none.

## current→calibrateFromPoints()[current calibrateFromPoints: ] YCurrent

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YCurrent **target calibrateFromPoints rawValues refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→describe()[current describe] YCurrent

Returns a short text that describes unambiguously the instance of the current sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the current sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## current→get_advertisedValue() current→advertisedValue()[current advertisedValue] YCurrent

Returns the current value of the current sensor (no more than 6 characters).

| | |
|---|---|
| `js` | function **get_advertisedValue**( ) |
| `nodejs` | function **get_advertisedValue**( ) |
| `php` | function **get_advertisedValue**( ) |
| `cpp` | string **get_advertisedValue**( ) |
| `m` | -(NSString*) advertisedValue |
| `pas` | function **get_advertisedValue**( ): string |
| `vb` | function **get_advertisedValue**( ) As String |
| `cs` | string **get_advertisedValue**( ) |
| `java` | String **get_advertisedValue**( ) |
| `py` | def **get_advertisedValue**( ) |
| `cmd` | YCurrent **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the current sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## current→get_currentRawValue() current→currentRawValue()[current currentRawValue] YCurrent

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YCurrent **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## current→get_currentValue() current→currentValue()[current currentValue] YCurrent

Returns the current measure for the current.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YCurrent **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current measure for the current

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## current→get_errorMessage() YCurrent
## current→errorMessage()[current errorMessage]

Returns the error message of the latest error with the current sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the current sensor object

## current→get_errorType() current→errorType() YCurrent

Returns the numerical error code of the latest error with the current sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the current sensor object

## current→get_friendlyName() current→friendlyName()[current friendlyName] YCurrent

Returns a global identifier of the current sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the current sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the current sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the current sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## current→get_functionDescriptor() current→functionDescriptor()[current functionDescriptor] YCurrent

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## current→get_functionId() current→functionId()[current functionId] YCurrent

Returns the hardware identifier of the current sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the current sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## current→get_hardwareId() current→hardwareId()[current hardwareId] YCurrent

Returns the unique hardware identifier of the current sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the current sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the current sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## current→get_highestValue() current→highestValue()[current highestValue] YCurrent

Returns the maximal value observed for the current.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YCurrent **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the current

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## current→get_logFrequency() current→logFrequency()[current logFrequency] YCurrent

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YCurrent **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## current→get_logicalName() current→logicalName()[current logicalName] YCurrent

Returns the logical name of the current sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YCurrent **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the current sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## current→get_lowestValue() current→lowestValue()[current lowestValue] YCurrent

Returns the minimal value observed for the current.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YCurrent **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the current

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## current→get_module() current→module()[current module] YCurrent

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## current→get_module_async() current→module_async() YCurrent

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## current→get_recordedData() YCurrent
## current→recordedData()[current recordedData: ]

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YCurrent **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## current→get_reportFrequency() current→reportFrequency()[current reportFrequency] YCurrent

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YCurrent **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## current→get_resolution() current→resolution()[current resolution] YCurrent

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YCurrent **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## current→get_unit()
## current→unit()[current unit]

**YCurrent**

Returns the measuring unit for the current.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YCurrent **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the current

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## current→get_userData() current→userData()[current userData] YCurrent

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## current→isOnline()[current isOnline] YCurrent

Checks if the current sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the current sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the current sensor.

**Returns :**

`true` if the current sensor can be reached, and `false` otherwise

## current→isOnline_async() YCurrent

Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

js function **isOnline_async**( **callback**, **context**)
nodejs function **isOnline_async**( **callback**, **context**)

If there is a cached value for the current sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## current→load()[current load: ] YCurrent

Preloads the current sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## current→loadCalibrationPoints()[current loadCalibrationPoints: ] YCurrent

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YCurrent **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→load_async() YCurrent

Preloads the current sensor cache with a specified validity duration (asynchronous version).

```
js     function load_async( msValidity, callback, context)
nodejs function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## current→nextCurrent()[current nextCurrent] YCurrent

Continues the enumeration of current sensors started using `yFirstCurrent()`.

| | |
|---|---|
| js | function **nextCurrent**( ) |
| nodejs | function **nextCurrent**( ) |
| php | function **nextCurrent**( ) |
| cpp | YCurrent * **nextCurrent**( ) |
| m | -(YCurrent*) **nextCurrent** |
| pas | function **nextCurrent**( ): TYCurrent |
| vb | function **nextCurrent**( ) As YCurrent |
| cs | YCurrent **nextCurrent**( ) |
| java | YCurrent **nextCurrent**( ) |
| py | def **nextCurrent**( ) |

**Returns :**

a pointer to a `YCurrent` object, corresponding to a current sensor currently online, or a `null` pointer if there are no more current sensors to enumerate.

## current→registerTimedReportCallback()[current registerTimedReportCallback: ] YCurrent

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YCurrentTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YCurrentTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYCurrentTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## current→registerValueCallback()[current registerValueCallback: ] YCurrent

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YCurrentValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YCurrentValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYCurrentValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## current→set_highestValue() current→setHighestValue()[current setHighestValue: ] YCurrent

Changes the recorded maximal value observed pour the current.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YCurrent **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed pour the current

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→set_logFrequency() current→setLogFrequency()[current setLogFrequency: ] YCurrent

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YCurrent **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→set_logicalName() current→setLogicalName()[current setLogicalName: ] YCurrent

Changes the logical name of the current sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YCurrent **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the 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 setLowestValue: ]

Changes the recorded minimal value observed pour the current.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YCurrent **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed pour the current

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→set_reportFrequency() current→setReportFrequency()[current setReportFrequency: ] YCurrent

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YCurrent **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→set_resolution() current→setResolution()[current setResolution: ] YCurrent

Changes the resolution of the measured values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YCurrent **target set_resolution newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## current→set_userData() current→setUserData()[current setUserData: ] YCurrent

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## current→wait_async() YCurrent

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.8. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_datalogger.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YDataLogger = yoctolib.YDataLogger; |
| php | require_once('yocto_datalogger.php'); |
| cpp | #include "yocto_datalogger.h" |
| m | #import "yocto_datalogger.h" |
| pas | uses yocto_datalogger; |
| vb | yocto_datalogger.vb |
| cs | yocto_datalogger.cs |
| java | import com.yoctopuce.YoctoAPI.YDataLogger; |
| py | from yocto_datalogger import * |

**Global functions**

**yFindDataLogger(func)**

Retrieves a data logger for a given identifier.

**yFirstDataLogger()**

Starts the enumeration of data loggers currently accessible.

**`YDataLogger` methods**

**datalogger→describe()**

Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**datalogger→forgetAllDataStreams()**

Clears the data logger memory and discards all recorded data streams.

**datalogger→get_advertisedValue()**

Returns the current value of the data logger (no more than 6 characters).

**datalogger→get_autoStart()**

Returns the default activation state of the data logger on power up.

**datalogger→get_currentRunIndex()**

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

**datalogger→get_dataSets()**

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

**datalogger→get_dataStreams(v)**

Builds a list of all data streams hold by the data logger (legacy method).

**datalogger→get_errorMessage()**

Returns the error message of the latest error with the data logger.

**datalogger→get_errorType()**

Returns the numerical error code of the latest error with the data logger.

**datalogger→get_friendlyName()**

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

**datalogger→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**datalogger→get_functionId()**

Returns the hardware identifier of the data logger, without reference to the module.

**datalogger→get_hardwareId()**

Returns the unique hardware identifier of the data logger in the form `SERIAL.FUNCTIONID`.

**datalogger→get_logicalName()**

Returns the logical name of the data logger.

**datalogger→get_module()**

Gets the `YModule` object for the device on which the function is located.

**datalogger→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**datalogger→get_recording()**

Returns the current activation state of the data logger.

**datalogger→get_timeUTC()**

Returns the Unix timestamp for current UTC time, if known.

**datalogger→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**datalogger→isOnline()**

Checks if the data logger is currently reachable, without raising any error.

**datalogger→isOnline_async(callback, context)**

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

**datalogger→load(msValidity)**

Preloads the data logger cache with a specified validity duration.

**datalogger→load_async(msValidity, callback, context)**

Preloads the data logger cache with a specified validity duration (asynchronous version).

**datalogger→nextDataLogger()**

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

**datalogger→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**datalogger→set_autoStart(newval)**

Changes the default activation state of the data logger on power up.

**datalogger→set_logicalName(newval)**

Changes the logical name of the data logger.

**datalogger→set_recording(newval)**

Changes the activation state of the data logger to start/stop recording data.

**datalogger→set_timeUTC(newval)**

Changes the current UTC time reference used for recorded data.

**datalogger→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**datalogger→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDataLogger.FindDataLogger() yFindDataLogger()yFindDataLogger() YDataLogger

Retrieves a data logger for a given identifier.

| | |
|---|---|
| js | function **yFindDataLogger**( **func**) |
| nodejs | function **FindDataLogger**( **func**) |
| php | function **yFindDataLogger**( **$func**) |
| cpp | YDataLogger* **yFindDataLogger**( string **func**) |
| m | +(YDataLogger*) **yFindDataLogger** : (NSString*) **func** |
| pas | function **yFindDataLogger**( **func**: string): TYDataLogger |
| vb | function **yFindDataLogger**( ByVal **func** As String) As YDataLogger |
| cs | YDataLogger **FindDataLogger**( string **func**) |
| java | YDataLogger **FindDataLogger**( String **func**) |
| py | def **FindDataLogger**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the data logger

**Returns :**

a `YDataLogger` object allowing you to drive the data logger.

## YDataLogger.FirstDataLogger() yFirstDataLogger()yFirstDataLogger() YDataLogger

Starts the enumeration of data loggers currently accessible.

| | |
|---|---|
| js | function **yFirstDataLogger**( ) |
| nodejs | function **FirstDataLogger**( ) |
| php | function **yFirstDataLogger**( ) |
| cpp | YDataLogger* **yFirstDataLogger**( ) |
| m | YDataLogger* **yFirstDataLogger**( ) |
| pas | function **yFirstDataLogger**( ): TYDataLogger |
| vb | function **yFirstDataLogger**( ) As YDataLogger |
| cs | YDataLogger **FirstDataLogger**( ) |
| java | YDataLogger **FirstDataLogger**( ) |
| py | def **FirstDataLogger**( ) |

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

**Returns :**

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

## datalogger→describe()[datalogger describe] YDataLogger

Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the data logger (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## datalogger→forgetAllDataStreams()[datalogger forgetAllDataStreams] YDataLogger

Clears the data logger memory and discards all recorded data streams.

| | |
|---|---|
| js | function **forgetAllDataStreams**( ) |
| nodejs | function **forgetAllDataStreams**( ) |
| php | function **forgetAllDataStreams**( ) |
| cpp | int **forgetAllDataStreams**( ) |
| m | -(int) **forgetAllDataStreams** |
| pas | function **forgetAllDataStreams**( ): LongInt |
| vb | function **forgetAllDataStreams**( ) As Integer |
| cs | int **forgetAllDataStreams**( ) |
| java | int **forgetAllDataStreams**( ) |
| py | def **forgetAllDataStreams**( ) |
| cmd | YDataLogger **target** **forgetAllDataStreams** |

This method also resets the current run index to zero.

**Returns :**
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→get_advertisedValue() datalogger→advertisedValue()[datalogger advertisedValue] YDataLogger

Returns the current value of the data logger (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YDataLogger **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the data logger (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## datalogger→get_autoStart() datalogger→autoStart()[datalogger autoStart]

**YDataLogger**

Returns the default activation state of the data logger on power up.

| | |
|---|---|
| js | function **get_autoStart**( ) |
| nodejs | function **get_autoStart**( ) |
| php | function **get_autoStart**( ) |
| cpp | Y_AUTOSTART_enum **get_autoStart**( ) |
| m | -(Y_AUTOSTART_enum) autoStart |
| pas | function **get_autoStart**( ): Integer |
| vb | function **get_autoStart**( ) As Integer |
| cs | int **get_autoStart**( ) |
| java | int **get_autoStart**( ) |
| py | def **get_autoStart**( ) |
| cmd | YDataLogger **target** **get_autoStart** |

**Returns :**

either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

On failure, throws an exception or returns `Y_AUTOSTART_INVALID`.

## datalogger→get_currentRunIndex() datalogger→currentRunIndex()[datalogger currentRunIndex] YDataLogger

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

| | |
|---|---|
| js | function **get_currentRunIndex**( ) |
| nodejs | function **get_currentRunIndex**( ) |
| php | function **get_currentRunIndex**( ) |
| cpp | int **get_currentRunIndex**( ) |
| m | -(int) currentRunIndex |
| pas | function **get_currentRunIndex**( ): LongInt |
| vb | function **get_currentRunIndex**( ) As Integer |
| cs | int **get_currentRunIndex**( ) |
| java | int **get_currentRunIndex**( ) |
| py | def **get_currentRunIndex**( ) |
| cmd | YDataLogger **target** **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 dataSets]

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

| | |
|---|---|
| js | function **get_dataSets**( ) |
| nodejs | function **get_dataSets**( ) |
| php | function **get_dataSets**( ) |
| cpp | vector<YDataSet> **get_dataSets**( ) |
| m | -(NSMutableArray*) dataSets |
| pas | function **get_dataSets**( ): TYDataSetArray |
| vb | function **get_dataSets**( ) As List |
| cs | List<YDataSet> **get_dataSets**( ) |
| java | ArrayList<YDataSet> **get_dataSets**( ) |
| py | def **get_dataSets**( ) |
| cmd | YDataLogger **target** **get_dataSets** |

This function only works if the device uses a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

**Returns :**
a list of YDataSet object.

On failure, throws an exception or returns an empty list.

## datalogger→get_dataStreams() datalogger→dataStreams()[datalogger dataStreams: ] YDataLogger

Builds a list of all data streams hold by the data logger (legacy method).

| | |
|---|---|
| js | function **get_dataStreams**( **v**) |
| nodejs | function **get_dataStreams**( **v**) |
| php | function **get_dataStreams**( **&$v**) |
| cpp | int **get_dataStreams**( ) |
| m | -(int) dataStreams : (NSArray**) **v** |
| pas | function **get_dataStreams**( **v**: Tlist): integer |
| vb | procedure **get_dataStreams**( ByVal **v** As List) |
| cs | int **get_dataStreams**( List<YDataStream> **v**) |
| java | int **get_dataStreams**( ArrayList<YDataStream> **v**) |
| py | def **get_dataStreams**( **v**) |

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

**Parameters :**

**v** an array of YDataStream objects to be filled in

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→get_errorMessage() datalogger→errorMessage()[datalogger errorMessage] YDataLogger

Returns the error message of the latest error with the data logger.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the data logger object

## datalogger→get_errorType() datalogger→errorType() YDataLogger

Returns the numerical error code of the latest error with the data logger.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the data logger object

## datalogger→get_friendlyName() datalogger→friendlyName()[datalogger friendlyName] YDataLogger

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the data logger using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## datalogger→get_functionDescriptor() datalogger→functionDescriptor()[datalogger functionDescriptor] YDataLogger

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## datalogger→get_functionId() datalogger→functionId()[datalogger functionId] YDataLogger

Returns the hardware identifier of the data logger, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the data logger (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## datalogger→get_hardwareId() datalogger→hardwareId()[datalogger hardwareId] YDataLogger

Returns the unique hardware identifier of the data logger in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the data logger (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## datalogger→get_logicalName() datalogger→logicalName()[datalogger logicalName] YDataLogger

Returns the logical name of the data logger.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YDataLogger **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the data logger. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## datalogger→get_module() datalogger→module()[datalogger module] YDataLogger

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## datalogger→get_module_async() datalogger→module_async() YDataLogger

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## datalogger→get_recording() datalogger→recording()[datalogger recording] YDataLogger

Returns the current activation state of the data logger.

| | |
|---|---|
| js | function **get_recording**( ) |
| nodejs | function **get_recording**( ) |
| php | function **get_recording**( ) |
| cpp | Y_RECORDING_enum **get_recording**( ) |
| m | -(Y_RECORDING_enum) recording |
| pas | function **get_recording**( ): Integer |
| vb | function **get_recording**( ) As Integer |
| cs | int **get_recording**( ) |
| java | int **get_recording**( ) |
| py | def **get_recording**( ) |
| cmd | YDataLogger **target** **get_recording** |

**Returns :**

either `Y_RECORDING_OFF` or `Y_RECORDING_ON`, according to the current activation state of the data logger

On failure, throws an exception or returns `Y_RECORDING_INVALID`.

## datalogger→get_timeUTC()
## datalogger→timeUTC()[datalogger timeUTC]

**YDataLogger**

Returns the Unix timestamp for current UTC time, if known.

| | |
|---|---|
| js | function **get_timeUTC**( ) |
| nodejs | function **get_timeUTC**( ) |
| php | function **get_timeUTC**( ) |
| cpp | s64 **get_timeUTC**( ) |
| m | -(s64) timeUTC |
| pas | function **get_timeUTC**( ): int64 |
| vb | function **get_timeUTC**( ) As Long |
| cs | long **get_timeUTC**( ) |
| java | long **get_timeUTC**( ) |
| py | def **get_timeUTC**( ) |
| cmd | YDataLogger **target** **get_timeUTC** |

**Returns :**

an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns `Y_TIMEUTC_INVALID`.

## datalogger→get_userData() datalogger→userData()[datalogger userData] — YDataLogger

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## datalogger→isOnline()[datalogger isOnline] YDataLogger

Checks if the data logger is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

**Returns :**
`true` if the data logger can be reached, and `false` otherwise

## datalogger→isOnline_async() YDataLogger

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## datalogger→load()[datalogger load: ] YDataLogger

Preloads the data logger cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## datalogger→load_async() YDataLogger

Preloads the data logger cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## datalogger→nextDataLogger()[datalogger nextDataLogger] YDataLogger

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

| | |
|---|---|
| js | function **nextDataLogger**( ) |
| nodejs | function **nextDataLogger**( ) |
| php | function **nextDataLogger**( ) |
| cpp | YDataLogger * **nextDataLogger**( ) |
| m | -(YDataLogger*) **nextDataLogger** |
| pas | function **nextDataLogger**( ): TYDataLogger |
| vb | function **nextDataLogger**( ) As YDataLogger |
| cs | YDataLogger **nextDataLogger**( ) |
| java | YDataLogger **nextDataLogger**( ) |
| py | def **nextDataLogger**( ) |

**Returns :**

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

## datalogger→registerValueCallback()[datalogger registerValueCallback: ] YDataLogger

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YDataLoggerValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YDataLoggerValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYDataLoggerValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## datalogger→set_autoStart() datalogger→setAutoStart()[datalogger setAutoStart: ] YDataLogger

Changes the default activation state of the data logger on power up.

| | |
|---|---|
| js | function **set_autoStart**( **newval**) |
| nodejs | function **set_autoStart**( **newval**) |
| php | function **set_autoStart**( **$newval**) |
| cpp | int **set_autoStart**( Y_AUTOSTART_enum **newval**) |
| m | -(int) setAutoStart : (Y_AUTOSTART_enum) **newval** |
| pas | function **set_autoStart**( **newval**: Integer): integer |
| vb | function **set_autoStart**( ByVal **newval** As Integer) As Integer |
| cs | int **set_autoStart**( int **newval**) |
| java | int **set_autoStart**( int **newval**) |
| py | def **set_autoStart**( **newval**) |
| cmd | YDataLogger **target set_autoStart newval** |

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→set_logicalName() datalogger→setLogicalName()[datalogger setLogicalName: ] YDataLogger

Changes the logical name of the data logger.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YDataLogger **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the data logger.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## datalogger→set_recording() datalogger→setRecording()[datalogger setRecording: ] YDataLogger

Changes the activation state of the data logger to start/stop recording data.

| | |
|---|---|
| js | function **set_recording**( **newval**) |
| nodejs | function **set_recording**( **newval**) |
| php | function **set_recording**( **$newval**) |
| cpp | int **set_recording**( Y_RECORDING_enum **newval**) |
| m | -(int) setRecording : (Y_RECORDING_enum) **newval** |
| pas | function **set_recording**( **newval**: Integer): integer |
| vb | function **set_recording**( ByVal **newval** As Integer) As Integer |
| cs | int **set_recording**( int **newval**) |
| java | int **set_recording**( int **newval**) |
| py | def **set_recording**( **newval**) |
| cmd | YDataLogger **target** **set_recording** **newval** |

**Parameters :**

**newval** either `Y_RECORDING_OFF` or `Y_RECORDING_ON`, according to the activation state of the data logger to start/stop recording data

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→set_timeUTC() datalogger→setTimeUTC()[datalogger setTimeUTC: ] YDataLogger

Changes the current UTC time reference used for recorded data.

| | |
|---|---|
| js | function **set_timeUTC**( **newval**) |
| nodejs | function **set_timeUTC**( **newval**) |
| php | function **set_timeUTC**( **$newval**) |
| cpp | int **set_timeUTC**( s64 **newval**) |
| m | -(int) setTimeUTC : (s64) **newval** |
| pas | function **set_timeUTC**( **newval**: int64): integer |
| vb | function **set_timeUTC**( ByVal **newval** As Long) As Integer |
| cs | int **set_timeUTC**( long **newval**) |
| java | int **set_timeUTC**( long **newval**) |
| py | def **set_timeUTC**( **newval**) |
| cmd | YDataLogger **target** **set_timeUTC** **newval** |

**Parameters :**

**newval** an integer corresponding to the current UTC time reference used for recorded data

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→set_userData() datalogger→setUserData()[datalogger setUserData: ] YDataLogger

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## datalogger→wait_async() YDataLogger

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.9. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_datalogger.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>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)**<br>Returns the average value of the measure observed at the specified time period. |
| **datarun→get_duration()**<br>Returns the duration (in seconds) of the data run. |
| **datarun→get_maxValue(measureName, pos)**<br>Returns the maximal value of the measure observed at the specified time period. |
| **datarun→get_measureNames()**<br>Returns the names of the measures recorded by the data logger. |
| **datarun→get_minValue(measureName, pos)**<br>Returns the minimal value of the measure observed at the specified time period. |
| **datarun→get_startTimeUTC()**<br>Returns the start time of the data run, relative to the Jan 1, 1970. |
| **datarun→get_valueCount()**<br>Returns the number of values accessible in this run, given the selected data samples interval. |
| **datarun→get_valueInterval()**<br>Returns the number of seconds covered by each value in this run. |
| **datarun→set_valueInterval(valueInterval)**<br>Changes the number of seconds covered by each value in this run. |

## datarun→get_averageValue() datarun→averageValue() YDataRun

Returns the average value of the measure observed at the specified time period.

| | |
|---|---|
| js | function **get_averageValue**( **measureName**, **pos**) |
| nodejs | function **get_averageValue**( **measureName**, **pos**) |
| php | function **get_averageValue**( **$measureName**, **$pos**) |
| java | double **get_averageValue**( String **measureName**, int **pos**) |
| py | def **get_averageValue**( **measureName**, **pos**) |

**Parameters :**

**measureName** the name of the desired measure (one of the names returned by `get_measureNames`)

**pos** the position index, between 0 and the value returned by `get_valueCount`

**Returns :**

a floating point number (the average value)

On failure, throws an exception or returns Y_AVERAGEVALUE_INVALID.

## datarun→get_duration()<br>datarun→duration() — YDataRun

Returns the duration (in seconds) of the data run.

| | |
|---|---|
| js | function **get_duration**( ) |
| nodejs | function **get_duration**( ) |
| php | function **get_duration**( ) |
| java | long **get_duration**( ) |
| py | def **get_duration**( ) |

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

**Returns :**

an unsigned number corresponding to the number of seconds between the beginning of the run (when the module was powered up) and the last recorded measure.

## datarun→get_maxValue() datarun→maxValue() YDataRun

Returns the maximal value of the measure observed at the specified time period.

| | |
|---|---|
| js | function **get_maxValue**( **measureName**, **pos**) |
| nodejs | function **get_maxValue**( **measureName**, **pos**) |
| php | function **get_maxValue**( **$measureName**, **$pos**) |
| java | double **get_maxValue**( String **measureName**, int **pos**) |
| py | def **get_maxValue**( **measureName**, **pos**) |

**Parameters :**

**measureName** the name of the desired measure (one of the names returned by `get_measureNames`)

**pos** the position index, between 0 and the value returned by `get_valueCount`

**Returns :**

a floating point number (the maximal value)

On failure, throws an exception or returns Y_MAXVALUE_INVALID.

## datarun→get_measureNames() datarun→measureNames() YDataRun

Returns the names of the measures recorded by the data logger.

`js` function **get_measureNames**( )
`nodejs` function **get_measureNames**( )
`php` function **get_measureNames**( )
`java` ArrayList<String> **get_measureNames**( )
`py` def **get_measureNames**( )

In most case, the measure names match the hardware identifier of the sensor that produced the data.

**Returns :**
a list of strings (the measure names) On failure, throws an exception or returns an empty array.

## datarun→get_minValue()
## datarun→minValue()

**YDataRun**

Returns the minimal value of the measure observed at the specified time period.

```
js      function get_minValue( measureName, pos)
nodejs  function get_minValue( measureName, pos)
php     function get_minValue( $measureName, $pos)
java    double get_minValue( String measureName, int pos)
py      def get_minValue( measureName, pos)
```

**Parameters :**

**measureName** the name of the desired measure (one of the names returned by `get_measureNames`)

**pos** the position index, between 0 and the value returned by `get_valueCount`

**Returns :**

a floating point number (the minimal value)

On failure, throws an exception or returns Y_MINVALUE_INVALID.

## datarun→get_startTimeUTC()<br>datarun→startTimeUTC() YDataRun

Returns the start time of the data run, relative to the Jan 1, 1970.

If the UTC time was not set in the datalogger at any time during the recording of this data run, and if this is not the current run, this method returns 0.

**Returns :**
an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data run (i.e. Unix time representation of the absolute time).

## datarun→get_valueCount() datarun→valueCount()

**YDataRun**

Returns the number of values accessible in this run, given the selected data samples interval.

| | |
|---|---|
| js | function **get_valueCount**( ) |
| nodejs | function **get_valueCount**( ) |
| php | function **get_valueCount**( ) |
| java | int **get_valueCount**( ) |
| py | def **get_valueCount**( ) |

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

**Returns :**

an unsigned number corresponding to the run duration divided by the samples interval.

## datarun→get_valueInterval()
## datarun→valueInterval()

**YDataRun**

Returns the number of seconds covered by each value in this run.

| | |
|---|---|
| js | function **get_valueInterval**( ) |
| nodejs | function **get_valueInterval**( ) |
| php | function **get_valueInterval**( ) |
| java | int **get_valueInterval**( ) |
| py | def **get_valueInterval**( ) |

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

**Returns :**

an unsigned number corresponding to a number of seconds covered by each data sample in the Run.

## datarun→set_valueInterval() datarun→setValueInterval() YDataRun

Changes the number of seconds covered by each value in this run.

| | |
|---|---|
| `js` | function **set_valueInterval**( **valueInterval**) |
| `nodejs` | function **set_valueInterval**( **valueInterval**) |
| `php` | function **set_valueInterval**( **$valueInterval**) |
| `java` | void **set_valueInterval**( int **valueInterval**) |
| `py` | def **set_valueInterval**( **valueInterval**) |

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

**Parameters :**

**valueInterval** an integer number of seconds.

**Returns :**

nothing

# 3.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instanciated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>var YModule = yoctolib.YModule; |
| php | require_once('yocto_api.php'); |
| cpp | #include "yocto_api.h" |
| m | #import "yocto_api.h" |
| pas | uses yocto_api; |
| vb | yocto_api.vb |
| cs | yocto_api.cs |
| java | import com.yoctopuce.YoctoAPI.YModule; |
| py | from yocto_api import * |

**`YDataSet` methods**

**dataset→get_endTimeUTC()**

Returns the end time of the dataset, relative to the Jan 1, 1970.

**dataset→get_functionId()**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

**dataset→get_hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

**dataset→get_measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

**dataset→get_preview()**

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

**dataset→get_progress()**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

**dataset→get_startTimeUTC()**

Returns the start time of the dataset, relative to the Jan 1, 1970.

**dataset→get_summary()**

Returns an YMeasure object which summarizes the whole DataSet.

**dataset→get_unit()**

Returns the measuring unit for the measured value.

**dataset→loadMore()**

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

**dataset→loadMore_async(callback, context)**

Loads the the next block of measures from the dataLogger asynchronously.

## dataset→get_endTimeUTC() dataset→endTimeUTC()[dataset endTimeUTC] — YDataSet

Returns the end time of the dataset, relative to the Jan 1, 1970.

| | |
|---|---|
| js | function **get_endTimeUTC**( ) |
| nodejs | function **get_endTimeUTC**( ) |
| php | function **get_endTimeUTC**( ) |
| cpp | s64 **get_endTimeUTC**( ) |
| m | -(s64) endTimeUTC |
| pas | function **get_endTimeUTC**( ): int64 |
| vb | function **get_endTimeUTC**( ) As Long |
| cs | long **get_endTimeUTC**( ) |
| java | long **get_endTimeUTC**( ) |
| py | def **get_endTimeUTC**( ) |

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

## dataset→get_functionId() dataset→functionId()[dataset functionId]

YDataSet

Returns the hardware identifier of the function that performed the measure, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| pas | function **get_functionId**( ): string |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `temperature1`.

**Returns :**

a string that identifies the function (ex: `temperature1`)

## dataset→get_hardwareId() dataset→hardwareId()[dataset hardwareId] YDataSet

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| pas | function **get_hardwareId**( ): string |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `THRMCPL1-123456.temperature1`)

**Returns :**

a string that uniquely identifies the function (ex: `THRMCPL1-123456.temperature1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## dataset→get_measures() dataset→measures()[dataset measures] YDataSet

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

| | |
|---|---|
| js | function **get_measures**( ) |
| nodejs | function **get_measures**( ) |
| php | function **get_measures**( ) |
| cpp | vector<YMeasure> **get_measures**( ) |
| m | -(NSMutableArray*) measures |
| pas | function **get_measures**( ): TYMeasureArray |
| vb | function **get_measures**( ) As List |
| cs | List<YMeasure> **get_measures**( ) |
| java | ArrayList<YMeasure> **get_measures**( ) |
| py | def **get_measures**( ) |

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call loadMore() several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

**Returns :**
a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

## dataset→get_preview() dataset→preview()[dataset preview] YDataSet

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

| | |
|---|---|
| js | function **get_preview**( ) |
| nodejs | function **get_preview**( ) |
| php | function **get_preview**( ) |
| cpp | vector<YMeasure> **get_preview**( ) |
| m | -(NSMutableArray*) preview |
| pas | function **get_preview**( ): TYMeasureArray |
| vb | function **get_preview**( ) As List |
| cs | List<YMeasure> **get_preview**( ) |
| java | ArrayList<YMeasure> **get_preview**( ) |
| py | def **get_preview**( ) |

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

**Returns :**

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

## dataset→get_progress() dataset→progress()[dataset progress]

**YDataSet**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

| | |
|---|---|
| js | function **get_progress**( ) |
| nodejs | function **get_progress**( ) |
| php | function **get_progress**( ) |
| cpp | int **get_progress**( ) |
| m | -(int) progress |
| pas | function **get_progress**( ): LongInt |
| vb | function **get_progress**( ) As Integer |
| cs | int **get_progress**( ) |
| java | int **get_progress**( ) |
| py | def **get_progress**( ) |

When the object is instanciated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

**Returns :**
an integer in the range 0 to 100 (percentage of completion).

## dataset→get_startTimeUTC() dataset→startTimeUTC()[dataset startTimeUTC] YDataSet

Returns the start time of the dataset, relative to the Jan 1, 1970.

| | |
|---|---|
| js | function **get_startTimeUTC**( ) |
| nodejs | function **get_startTimeUTC**( ) |
| php | function **get_startTimeUTC**( ) |
| cpp | s64 **get_startTimeUTC**( ) |
| m | -(s64) startTimeUTC |
| pas | function **get_startTimeUTC**( ): int64 |
| vb | function **get_startTimeUTC**( ) As Long |
| cs | long **get_startTimeUTC**( ) |
| java | long **get_startTimeUTC**( ) |
| py | def **get_startTimeUTC**( ) |

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

## dataset→get_summary() dataset→summary()[dataset summary] YDataSet

Returns an YMeasure object which summarizes the whole DataSet.

| | |
|---|---|
| js | function **get_summary**( ) |
| nodejs | function **get_summary**( ) |
| php | function **get_summary**( ) |
| cpp | YMeasure **get_summary**( ) |
| m | -(YMeasure*) summary |
| pas | function **get_summary**( ): TYMeasure |
| vb | function **get_summary**( ) As YMeasure |
| cs | YMeasure **get_summary**( ) |
| java | YMeasure **get_summary**( ) |
| py | def **get_summary**( ) |

In includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

**Returns :**
an YMeasure object

## dataset→get_unit() dataset→unit()[dataset unit] — YDataSet

Returns the measuring unit for the measured value.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |

**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.

| | |
|---|---|
| js | function **loadMore**( ) |
| nodejs | function **loadMore**( ) |
| php | function **loadMore**( ) |
| cpp | int **loadMore**( ) |
| m | -(int) **loadMore** |
| pas | function **loadMore**( ): LongInt |
| vb | function **loadMore**( ) As Integer |
| cs | int **loadMore**( ) |
| java | int **loadMore**( ) |
| py | def **loadMore**( ) |

**Returns :**
an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

## dataset→loadMore_async() YDataSet

Loads the the next block of measures from the dataLogger asynchronously.

| | |
|---|---|
| js | function **loadMore_async**( **callback**, **context**) |
| nodejs | function **loadMore_async**( **callback**, **context**) |

**Parameters :**

**callback** callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YDataSet object whose loadMore_async was invoked - the load result: either the progress indicator (0...100), or a negative error code in case of failure.

**context** user-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.11. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>var YModule = yoctolib.YModule; |
| php | require_once('yocto_api.php'); |
| cpp | #include "yocto_api.h" |
| m | #import "yocto_api.h" |
| pas | uses yocto_api; |
| vb | yocto_api.vb |
| cs | yocto_api.cs |
| java | import com.yoctopuce.YoctoAPI.YModule; |
| py | from yocto_api import * |

| **`YDataStream` methods** |
|---|
| **datastream→get_averageValue()**<br>Returns the average of all measures observed within this stream. |
| **datastream→get_columnCount()**<br>Returns the number of data columns present in this stream. |
| **datastream→get_columnNames()**<br>Returns the title (or meaning) of each data column present in this stream. |
| **datastream→get_data(row, col)**<br>Returns a single measure from the data stream, specified by its row and column index. |
| **datastream→get_dataRows()**<br>Returns the whole data set contained in the stream, as a bidimensional table of numbers. |
| **datastream→get_dataSamplesIntervalMs()**<br>Returns the number of milliseconds between two consecutive rows of this data stream. |
| **datastream→get_duration()**<br>Returns the approximate duration of this stream, in seconds. |
| **datastream→get_maxValue()**<br>Returns the largest measure observed within this stream. |
| **datastream→get_minValue()**<br>Returns the smallest measure observed within this stream. |
| **datastream→get_rowCount()**<br>Returns the number of data rows present in this stream. |
| **datastream→get_runIndex()**<br>Returns the run index of the data stream. |
| **datastream→get_startTime()**<br>Returns the relative start time of the data stream, measured in seconds. |
| **datastream→get_startTimeUTC()** |

Returns the start time of the data stream, relative to the Jan 1, 1970.

## datastream→get_averageValue() datastream→averageValue()[datastream averageValue] YDataStream

Returns the average of all measures observed within this stream.

| | |
|---|---|
| js | function **get_averageValue**( ) |
| nodejs | function **get_averageValue**( ) |
| php | function **get_averageValue**( ) |
| cpp | double **get_averageValue**( ) |
| m | -(double) averageValue |
| pas | function **get_averageValue**( ): double |
| vb | function **get_averageValue**( ) As Double |
| cs | double **get_averageValue**( ) |
| java | double **get_averageValue**( ) |
| py | def **get_averageValue**( ) |

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

**Returns :**

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

## datastream→get_columnCount() datastream→columnCount()[datastream columnCount] YDataStream

Returns the number of data columns present in this stream.

| | |
|---|---|
| js | function **get_columnCount**( ) |
| nodejs | function **get_columnCount**( ) |
| php | function **get_columnCount**( ) |
| cpp | int **get_columnCount**( ) |
| m | -(int) columnCount |
| pas | function **get_columnCount**( ): LongInt |
| vb | function **get_columnCount**( ) As Integer |
| cs | int **get_columnCount**( ) |
| java | int **get_columnCount**( ) |
| py | def **get_columnCount**( ) |

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

**Returns :**
an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

## datastream→get_columnNames() datastream→columnNames()[datastream columnNames] YDataStream

Returns the title (or meaning) of each data column present in this stream.

| | |
|---|---|
| js | function **get_columnNames**( ) |
| nodejs | function **get_columnNames**( ) |
| php | function **get_columnNames**( ) |
| cpp | vector<string> **get_columnNames**( ) |
| m | -(NSMutableArray*) columnNames |
| pas | function **get_columnNames**( ): TStringArray |
| vb | function **get_columnNames**( ) As List |
| cs | List<string> **get_columnNames**( ) |
| java | ArrayList<String> **get_columnNames**( ) |
| py | def **get_columnNames**( ) |

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes _min, _avg and _max respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

**Returns :**

a list containing as many strings as there are columns in the data stream.

On failure, throws an exception or returns an empty array.

## datastream→get_data() datastream→data()[datastream data: ] YDataStream

Returns a single measure from the data stream, specified by its row and column index.

| | |
|---|---|
| js | function **get_data**( **row**, **col**) |
| nodejs | function **get_data**( **row**, **col**) |
| php | function **get_data**( **$row**, **$col**) |
| cpp | double **get_data**( int **row**, int **col**) |
| m | -(double) data : (int) **row**<br>: (int) **col** |
| pas | function **get_data**( **row**: LongInt, **col**: LongInt): double |
| vb | function **get_data**( ) As Double |
| cs | double **get_data**( int **row**, int **col**) |
| java | double **get_data**( int **row**, int **col**) |
| py | def **get_data**( **row**, **col**) |

The meaning of the values present in each column can be obtained using the method get_columnNames().

This method fetches the whole data stream from the device, if not yet done.

**Parameters :**

**row** row index

**col** column index

**Returns :**

a floating-point number

On failure, throws an exception or returns Y_DATA_INVALID.

## datastream→get_dataRows() datastream→dataRows()[datastream dataRows] YDataStream

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

| | |
|---|---|
| js | function **get_dataRows**( ) |
| nodejs | function **get_dataRows**( ) |
| php | function **get_dataRows**( ) |
| cpp | vector< vector<double> > **get_dataRows**( ) |
| m | -(NSMutableArray*) dataRows |
| pas | function **get_dataRows**( ): TDoubleArrayArray |
| vb | function **get_dataRows**( ) As List |
| cs | List<List<double>> **get_dataRows**( ) |
| java | ArrayList<ArrayList<Double>> **get_dataRows**( ) |
| py | def **get_dataRows**( ) |

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

**Returns :**

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.

## datastream→get_dataSamplesIntervalMs() datastream→dataSamplesIntervalMs()[datastream dataSamplesIntervalMs] YDataStream

Returns the number of milliseconds between two consecutive rows of this data stream.

| | |
|---|---|
| js | function **get_dataSamplesIntervalMs**( ) |
| nodejs | function **get_dataSamplesIntervalMs**( ) |
| php | function **get_dataSamplesIntervalMs**( ) |
| cpp | int **get_dataSamplesIntervalMs**( ) |
| m | -(int) dataSamplesIntervalMs |
| pas | function **get_dataSamplesIntervalMs**( ): LongInt |
| vb | function **get_dataSamplesIntervalMs**( ) As Integer |
| cs | int **get_dataSamplesIntervalMs**( ) |
| java | int **get_dataSamplesIntervalMs**( ) |
| py | def **get_dataSamplesIntervalMs**( ) |

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

**Returns :**

an unsigned number corresponding to a number of milliseconds.

## datastream→get_duration() datastream→duration()[datastream duration] YDataStream

Returns the approximate duration of this stream, in seconds.

| | |
|---|---|
| js | function **get_duration**( ) |
| nodejs | function **get_duration**( ) |
| php | function **get_duration**( ) |
| cpp | int **get_duration**( ) |
| m | -(int) duration |
| pas | function **get_duration**( ): LongInt |
| vb | function **get_duration**( ) As Integer |
| cs | int **get_duration**( ) |
| java | int **get_duration**( ) |
| py | def **get_duration**( ) |

**Returns :**
the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

## datastream→get_maxValue() datastream→maxValue()[datastream maxValue] YDataStream

Returns the largest measure observed within this stream.

| | |
|---|---|
| js | function **get_maxValue**( ) |
| nodejs | function **get_maxValue**( ) |
| php | function **get_maxValue**( ) |
| cpp | double **get_maxValue**( ) |
| m | -(double) maxValue |
| pas | function **get_maxValue**( ): double |
| vb | function **get_maxValue**( ) As Double |
| cs | double **get_maxValue**( ) |
| java | double **get_maxValue**( ) |
| py | def **get_maxValue**( ) |

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

**Returns :**

a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

## datastream→get_minValue() datastream→minValue()[datastream minValue]

**YDataStream**

Returns the smallest measure observed within this stream.

| | |
|---|---|
| js | function **get_minValue**( ) |
| nodejs | function **get_minValue**( ) |
| php | function **get_minValue**( ) |
| cpp | double **get_minValue**( ) |
| m | -(double) minValue |
| pas | function **get_minValue**( ): double |
| vb | function **get_minValue**( ) As Double |
| cs | double **get_minValue**( ) |
| java | double **get_minValue**( ) |
| py | def **get_minValue**( ) |

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

**Returns :**

a floating-point number corresponding to the smallest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

## datastream→get_rowCount() datastream→rowCount()[datastream rowCount] YDataStream

Returns the number of data rows present in this stream.

| | |
|---|---|
| js | function **get_rowCount**( ) |
| nodejs | function **get_rowCount**( ) |
| php | function **get_rowCount**( ) |
| cpp | int **get_rowCount**( ) |
| m | -(int) rowCount |
| pas | function **get_rowCount**( ): LongInt |
| vb | function **get_rowCount**( ) As Integer |
| cs | int **get_rowCount**( ) |
| java | int **get_rowCount**( ) |
| py | def **get_rowCount**( ) |

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

**Returns :**

an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

## datastream→get_runIndex() datastream→runIndex()[datastream runIndex] YDataStream

Returns the run index of the data stream.

| | |
|---|---|
| js | function **get_runIndex**( ) |
| nodejs | function **get_runIndex**( ) |
| php | function **get_runIndex**( ) |
| cpp | int **get_runIndex**( ) |
| m | -(int) runIndex |
| pas | function **get_runIndex**( ): LongInt |
| vb | function **get_runIndex**( ) As Integer |
| cs | int **get_runIndex**( ) |
| java | int **get_runIndex**( ) |
| py | def **get_runIndex**( ) |

A run can be made of multiple datastreams, for different time intervals.

**Returns :**
an unsigned number corresponding to the run index.

## datastream→get_startTime() datastream→startTime()[datastream startTime] YDataStream

Returns the relative start time of the data stream, measured in seconds.

| | |
|---|---|
| js | function **get_startTime**( ) |
| nodejs | function **get_startTime**( ) |
| php | function **get_startTime**( ) |
| cpp | int **get_startTime**( ) |
| m | -(int) startTime |
| pas | function **get_startTime**( ): LongInt |
| vb | function **get_startTime**( ) As Integer |
| cs | int **get_startTime**( ) |
| java | int **get_startTime**( ) |
| py | def **get_startTime**( ) |

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `get_startTimeUTC()`.

**Returns :**
an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

## datastream→get_startTimeUTC() datastream→startTimeUTC()[datastream startTimeUTC] YDataStream

Returns the start time of the data stream, relative to the Jan 1, 1970.

| | |
|---|---|
| js | function **get_startTimeUTC**( ) |
| nodejs | function **get_startTimeUTC**( ) |
| php | function **get_startTimeUTC**( ) |
| cpp | s64 **get_startTimeUTC**( ) |
| m | -(s64) startTimeUTC |
| pas | function **get_startTimeUTC**( ): int64 |
| vb | function **get_startTimeUTC**( ) As Long |
| cs | long **get_startTimeUTC**( ) |
| java | long **get_startTimeUTC**( ) |
| py | def **get_startTimeUTC**( ) |

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data stream (i.e. Unix time representation of the absolute time).

# 3.12. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_digitalio.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YDigitalIO = yoctolib.YDigitalIO; |
| php | require_once('yocto_digitalio.php'); |
| cpp | #include "yocto_digitalio.h" |
| m | #import "yocto_digitalio.h" |
| pas | uses yocto_digitalio; |
| vb | yocto_digitalio.vb |
| cs | yocto_digitalio.cs |
| java | import com.yoctopuce.YoctoAPI.YDigitalIO; |
| py | from yocto_digitalio import * |

**Global functions**

**yFindDigitalIO(func)**

Retrieves a digital IO port for a given identifier.

**yFirstDigitalIO()**

Starts the enumeration of digital IO ports currently accessible.

**`YDigitalIO` methods**

**digitalio→delayedPulse(bitno, ms_delay, ms_duration)**

Schedules a pulse on a single bit for a specified duration.

**digitalio→describe()**

Returns a short text that describes unambiguously the instance of the digital IO port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**digitalio→get_advertisedValue()**

Returns the current value of the digital IO port (no more than 6 characters).

**digitalio→get_bitDirection(bitno)**

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

**digitalio→get_bitOpenDrain(bitno)**

Returns the type of electrical interface of a single bit from the I/O port.

**digitalio→get_bitPolarity(bitno)**

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

**digitalio→get_bitState(bitno)**

Returns the state of a single bit of the I/O port.

**digitalio→get_errorMessage()**

Returns the error message of the latest error with the digital IO port.

**digitalio→get_errorType()**

Returns the numerical error code of the latest error with the digital IO port.

**digitalio→get_friendlyName()**

Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

**digitalio→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**digitalio→get_functionId()**
Returns the hardware identifier of the digital IO port, without reference to the module.

**digitalio→get_hardwareId()**
Returns the unique hardware identifier of the digital IO port in the form `SERIAL.FUNCTIONID`.

**digitalio→get_logicalName()**
Returns the logical name of the digital IO port.

**digitalio→get_module()**
Gets the `YModule` object for the device on which the function is located.

**digitalio→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**digitalio→get_outputVoltage()**
Returns the voltage source used to drive output bits.

**digitalio→get_portDirection()**
Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

**digitalio→get_portOpenDrain()**
Returns the electrical interface for each bit of the port.

**digitalio→get_portPolarity()**
Returns the polarity of all the bits of the port.

**digitalio→get_portSize()**
Returns the number of bits implemented in the I/O port.

**digitalio→get_portState()**
Returns the digital IO port state: bit 0 represents input 0, and so on.

**digitalio→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**digitalio→isOnline()**
Checks if the digital IO port is currently reachable, without raising any error.

**digitalio→isOnline_async(callback, context)**
Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

**digitalio→load(msValidity)**
Preloads the digital IO port cache with a specified validity duration.

**digitalio→load_async(msValidity, callback, context)**
Preloads the digital IO port cache with a specified validity duration (asynchronous version).

**digitalio→nextDigitalIO()**
Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

**digitalio→pulse(bitno, ms_duration)**
Triggers a pulse on a single bit for a specified duration.

**digitalio→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**digitalio→set_bitDirection(bitno, bitdirection)**
Changes the direction of a single bit from the I/O port.

**digitalio→set_bitOpenDrain(bitno, opendrain)**
Changes the electrical interface of a single bit from the I/O port.

**digitalio→set_bitPolarity(bitno, bitpolarity)**

Changes the polarity of a single bit from the I/O port.

**digitalio→set_bitState(bitno, bitstate)**

Sets a single bit of the I/O port.

**digitalio→set_logicalName(newval)**

Changes the logical name of the digital IO port.

**digitalio→set_outputVoltage(newval)**

Changes the voltage source used to drive output bits.

**digitalio→set_portDirection(newval)**

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

**digitalio→set_portOpenDrain(newval)**

Changes the electrical interface for each bit of the port.

**digitalio→set_portPolarity(newval)**

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

**digitalio→set_portState(newval)**

Changes the digital IO port state: bit 0 represents input 0, and so on.

**digitalio→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**digitalio→toggle_bitState(bitno)**

Reverts a single bit of the I/O port.

**digitalio→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDigitalIO.FindDigitalIO() yFindDigitalIO()yFindDigitalIO() YDigitalIO

Retrieves a digital IO port for a given identifier.

| | |
|---|---|
| js | function **yFindDigitalIO**( **func**) |
| nodejs | function **FindDigitalIO**( **func**) |
| php | function **yFindDigitalIO**( **$func**) |
| cpp | YDigitalIO* **yFindDigitalIO**( const string& **func**) |
| m | YDigitalIO* **yFindDigitalIO**( NSString* **func**) |
| pas | function **yFindDigitalIO**( **func**: string): TYDigitalIO |
| vb | function **yFindDigitalIO**( ByVal **func** As String) As YDigitalIO |
| cs | YDigitalIO **FindDigitalIO**( string **func**) |
| java | YDigitalIO **FindDigitalIO**( String **func**) |
| py | def **FindDigitalIO**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the digital IO port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDigitalIO.isOnline()` to test if the digital IO port is indeed online at a given time. In case of ambiguity when looking for a digital IO port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the digital IO port

**Returns :**

a `YDigitalIO` object allowing you to drive the digital IO port.

## YDigitalIO.FirstDigitalIO() yFirstDigitalIO()yFirstDigitalIO() YDigitalIO

Starts the enumeration of digital IO ports currently accessible.

| | |
|---|---|
| js | function **yFirstDigitalIO**( ) |
| nodejs | function **FirstDigitalIO**( ) |
| php | function **yFirstDigitalIO**( ) |
| cpp | YDigitalIO* **yFirstDigitalIO**( ) |
| m | YDigitalIO* **yFirstDigitalIO**( ) |
| pas | function **yFirstDigitalIO**( ): TYDigitalIO |
| vb | function **yFirstDigitalIO**( ) As YDigitalIO |
| cs | YDigitalIO **FirstDigitalIO**( ) |
| java | YDigitalIO **FirstDigitalIO**( ) |
| py | def **FirstDigitalIO**( ) |

Use the method `YDigitalIO.nextDigitalIO()` to iterate on next digital IO ports.

**Returns :**

a pointer to a `YDigitalIO` object, corresponding to the first digital IO port currently online, or a `null` pointer if there are none.

## digitalio→delayedPulse()[digitalio delayedPulse: ] YDigitalIO

Schedules a pulse on a single bit for a specified duration.

| | |
|---|---|
| js | function **delayedPulse**( **bitno**, **ms_delay**, **ms_duration**) |
| nodejs | function **delayedPulse**( **bitno**, **ms_delay**, **ms_duration**) |
| php | function **delayedPulse**( **$bitno**, **$ms_delay**, **$ms_duration**) |
| cpp | int **delayedPulse**( int **bitno**, int **ms_delay**, int **ms_duration**) |
| m | -(int) **delayedPulse** : (int) **bitno** : (int) **ms_delay** : (int) **ms_duration** |
| pas | function **delayedPulse**( **bitno**: LongInt, **ms_delay**: LongInt, **ms_duration**: LongInt): LongInt |
| vb | function **delayedPulse**( ) As Integer |
| cs | int **delayedPulse**( int **bitno**, int **ms_delay**, int **ms_duration**) |
| java | int **delayedPulse**( int **bitno**, int **ms_delay**, int **ms_duration**) |
| py | def **delayedPulse**( **bitno**, **ms_delay**, **ms_duration**) |
| cmd | YDigitalIO **target** **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`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the digital IO port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## digitalio→get_advertisedValue() digitalio→advertisedValue()[digitalio advertisedValue] YDigitalIO

Returns the current value of the digital IO port (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YDigitalIO **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the digital IO port (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## digitalio→get_bitDirection() digitalio→bitDirection()[digitalio bitDirection: ] YDigitalIO

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

| | |
|---|---|
| js | function **get_bitDirection**( **bitno**) |
| nodejs | function **get_bitDirection**( **bitno**) |
| php | function **get_bitDirection**( **$bitno**) |
| cpp | int **get_bitDirection**( int **bitno**) |
| m | -(int) bitDirection : (int) **bitno** |
| pas | function **get_bitDirection**( **bitno**: LongInt): LongInt |
| vb | function **get_bitDirection**( ) As Integer |
| cs | int **get_bitDirection**( int **bitno**) |
| java | int **get_bitDirection**( int **bitno**) |
| py | def **get_bitDirection**( **bitno**) |
| cmd | YDigitalIO **target get_bitDirection bitno** |

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→get_bitOpenDrain() digitalio→bitOpenDrain()[digitalio bitOpenDrain: ] YDigitalIO

Returns the type of electrical interface of a single bit from the I/O port.

| | |
|---|---|
| js | function **get_bitOpenDrain**( **bitno**) |
| nodejs | function **get_bitOpenDrain**( **bitno**) |
| php | function **get_bitOpenDrain**( **$bitno**) |
| cpp | int **get_bitOpenDrain**( int **bitno**) |
| m | -(int) bitOpenDrain : (int) **bitno** |
| pas | function **get_bitOpenDrain**( **bitno**: LongInt): LongInt |
| vb | function **get_bitOpenDrain**( ) As Integer |
| cs | int **get_bitOpenDrain**( int **bitno**) |
| java | int **get_bitOpenDrain**( int **bitno**) |
| py | def **get_bitOpenDrain**( **bitno**) |
| cmd | YDigitalIO **target 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()<br>digitalio→bitPolarity()[digitalio bitPolarity: ] YDigitalIO

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

| | |
|---|---|
| js | function **get_bitPolarity**( **bitno**) |
| nodejs | function **get_bitPolarity**( **bitno**) |
| php | function **get_bitPolarity**( **$bitno**) |
| cpp | int **get_bitPolarity**( int **bitno**) |
| m | -(int) bitPolarity : (int) **bitno** |
| pas | function **get_bitPolarity**( **bitno**: LongInt): LongInt |
| vb | function **get_bitPolarity**( ) As Integer |
| cs | int **get_bitPolarity**( int **bitno**) |
| java | int **get_bitPolarity**( int **bitno**) |
| py | def **get_bitPolarity**( **bitno**) |
| cmd | YDigitalIO **target** **get_bitPolarity** **bitno** |

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→get_bitState() digitalio→bitState()[digitalio bitState: ] YDigitalIO

Returns the state of a single bit of the I/O port.

| | |
|---|---|
| js | function **get_bitState**( **bitno**) |
| nodejs | function **get_bitState**( **bitno**) |
| php | function **get_bitState**( **$bitno**) |
| cpp | int **get_bitState**( int **bitno**) |
| m | -(int) bitState : (int) **bitno** |
| pas | function **get_bitState**( **bitno**: LongInt): LongInt |
| vb | function **get_bitState**( ) As Integer |
| cs | int **get_bitState**( int **bitno**) |
| java | int **get_bitState**( int **bitno**) |
| py | def **get_bitState**( **bitno**) |
| cmd | YDigitalIO **target** **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 errorMessage]

Returns the error message of the latest error with the digital IO port.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the digital IO port object

## digitalio→get_errorType() digitalio→errorType() YDigitalIO

Returns the numerical error code of the latest error with the digital IO port.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the digital IO port object

## digitalio→get_friendlyName() digitalio→friendlyName()[digitalio friendlyName] YDigitalIO

Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the digital IO port if they are defined, otherwise the serial number of the module and the hardware identifier of the digital IO port (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the digital IO port using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## digitalio→get_functionDescriptor() digitalio→functionDescriptor()[digitalio functionDescriptor] YDigitalIO

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## digitalio→get_functionId() digitalio→functionId()[digitalio functionId] YDigitalIO

Returns the hardware identifier of the digital IO port, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the digital IO port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## digitalio→get_hardwareId() digitalio→hardwareId()[digitalio hardwareId] YDigitalIO

Returns the unique hardware identifier of the digital IO port in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| `js` | function **get_hardwareId**( ) |
| `nodejs` | function **get_hardwareId**( ) |
| `php` | function **get_hardwareId**( ) |
| `cpp` | string **get_hardwareId**( ) |
| `m` | -(NSString*) hardwareId |
| `vb` | function **get_hardwareId**( ) As String |
| `cs` | string **get_hardwareId**( ) |
| `java` | String **get_hardwareId**( ) |
| `py` | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the digital IO port. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the digital IO port (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## digitalio→get_logicalName() digitalio→logicalName()[digitalio logicalName] YDigitalIO

Returns the logical name of the digital IO port.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YDigitalIO **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the digital IO port. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## digitalio→get_module() digitalio→module()[digitalio module] YDigitalIO

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## digitalio→get_module_async() digitalio→module_async() YDigitalIO

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## digitalio→get_outputVoltage() digitalio→outputVoltage()[digitalio outputVoltage] YDigitalIO

Returns the voltage source used to drive output bits.

| | |
|---|---|
| js | function **get_outputVoltage**( ) |
| nodejs | function **get_outputVoltage**( ) |
| php | function **get_outputVoltage**( ) |
| cpp | Y_OUTPUTVOLTAGE_enum **get_outputVoltage**( ) |
| m | -(Y_OUTPUTVOLTAGE_enum) outputVoltage |
| pas | function **get_outputVoltage**( ): Integer |
| vb | function **get_outputVoltage**( ) As Integer |
| cs | int **get_outputVoltage**( ) |
| java | int **get_outputVoltage**( ) |
| py | def **get_outputVoltage**( ) |
| cmd | YDigitalIO **target** **get_outputVoltage** |

**Returns :**

a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

On failure, throws an exception or returns `Y_OUTPUTVOLTAGE_INVALID`.

## digitalio→get_portDirection() digitalio→portDirection()[digitalio portDirection] YDigitalIO

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

| | |
|---|---|
| js | function **get_portDirection**( ) |
| nodejs | function **get_portDirection**( ) |
| php | function **get_portDirection**( ) |
| cpp | int **get_portDirection**( ) |
| m | -(int) portDirection |
| pas | function **get_portDirection**( ): LongInt |
| vb | function **get_portDirection**( ) As Integer |
| cs | int **get_portDirection**( ) |
| java | int **get_portDirection**( ) |
| py | def **get_portDirection**( ) |
| cmd | YDigitalIO **target** **get_portDirection** |

**Returns :**

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

On failure, throws an exception or returns `Y_PORTDIRECTION_INVALID`.

## digitalio→get_portOpenDrain() digitalio→portOpenDrain()[digitalio portOpenDrain] YDigitalIO

Returns the electrical interface for each bit of the port.

| | |
|---|---|
| js | function **get_portOpenDrain**( ) |
| nodejs | function **get_portOpenDrain**( ) |
| php | function **get_portOpenDrain**( ) |
| cpp | int **get_portOpenDrain**( ) |
| m | -(int) portOpenDrain |
| pas | function **get_portOpenDrain**( ): LongInt |
| vb | function **get_portOpenDrain**( ) As Integer |
| cs | int **get_portOpenDrain**( ) |
| java | int **get_portOpenDrain**( ) |
| py | def **get_portOpenDrain**( ) |
| cmd | YDigitalIO **target** **get_portOpenDrain** |

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

**Returns :**
an integer corresponding to the electrical interface for each bit of the port

On failure, throws an exception or returns `Y_PORTOPENDRAIN_INVALID`.

## digitalio→get_portPolarity() digitalio→portPolarity()[digitalio portPolarity] YDigitalIO

Returns the polarity of all the bits of the port.

| | |
|---|---|
| js | function **get_portPolarity**( ) |
| nodejs | function **get_portPolarity**( ) |
| php | function **get_portPolarity**( ) |
| cpp | int **get_portPolarity**( ) |
| m | -(int) portPolarity |
| pas | function **get_portPolarity**( ): LongInt |
| vb | function **get_portPolarity**( ) As Integer |
| cs | int **get_portPolarity**( ) |
| java | int **get_portPolarity**( ) |
| py | def **get_portPolarity**( ) |
| cmd | YDigitalIO **target** **get_portPolarity** |

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

**Returns :**

an integer corresponding to the polarity of all the bits of the port

On failure, throws an exception or returns `Y_PORTPOLARITY_INVALID`.

## digitalio→get_portSize() digitalio→portSize()[digitalio portSize]

**YDigitalIO**

Returns the number of bits implemented in the I/O port.

| | |
|---|---|
| js | function **get_portSize**( ) |
| nodejs | function **get_portSize**( ) |
| php | function **get_portSize**( ) |
| cpp | int **get_portSize**( ) |
| m | -(int) portSize |
| pas | function **get_portSize**( ): LongInt |
| vb | function **get_portSize**( ) As Integer |
| cs | int **get_portSize**( ) |
| java | int **get_portSize**( ) |
| py | def **get_portSize**( ) |
| cmd | YDigitalIO **target** **get_portSize** |

**Returns :**

an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns `Y_PORTSIZE_INVALID`.

## digitalio→get_portState() digitalio→portState()[digitalio portState] YDigitalIO

Returns the digital IO port state: bit 0 represents input 0, and so on.

| | |
|---|---|
| js | function **get_portState**( ) |
| nodejs | function **get_portState**( ) |
| php | function **get_portState**( ) |
| cpp | int **get_portState**( ) |
| m | -(int) portState |
| pas | function **get_portState**( ): LongInt |
| vb | function **get_portState**( ) As Integer |
| cs | int **get_portState**( ) |
| java | int **get_portState**( ) |
| py | def **get_portState**( ) |
| cmd | YDigitalIO **target** **get_portState** |

**Returns :**
an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

On failure, throws an exception or returns `Y_PORTSTATE_INVALID`.

## digitalio→get_userData() digitalio→userData()[digitalio userData] YDigitalIO

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| `js` | function **get_userData**( ) |
| `nodejs` | function **get_userData**( ) |
| `php` | function **get_userData**( ) |
| `cpp` | void * **get_userData**( ) |
| `m` | -(void*) userData |
| `pas` | function **get_userData**( ): Tobject |
| `vb` | function **get_userData**( ) As Object |
| `cs` | object **get_userData**( ) |
| `java` | Object **get_userData**( ) |
| `py` | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## digitalio→isOnline()[digitalio isOnline] YDigitalIO

Checks if the digital IO port is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the digital IO port.

**Returns :**
`true` if the digital IO port can be reached, and `false` otherwise

## digitalio→isOnline_async() YDigitalIO

Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## digitalio→load()[digitalio load: ] YDigitalIO

Preloads the digital IO port cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## digitalio→load_async() YDigitalIO

Preloads the digital IO port cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## digitalio→nextDigitalIO()[digitalio nextDigitalIO] YDigitalIO

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

| | |
|---|---|
| js | function **nextDigitalIO**( ) |
| nodejs | function **nextDigitalIO**( ) |
| php | function **nextDigitalIO**( ) |
| cpp | YDigitalIO * **nextDigitalIO**( ) |
| m | -(YDigitalIO*) **nextDigitalIO** |
| pas | function **nextDigitalIO**( ): TYDigitalIO |
| vb | function **nextDigitalIO**( ) As YDigitalIO |
| cs | YDigitalIO **nextDigitalIO**( ) |
| java | YDigitalIO **nextDigitalIO**( ) |
| py | def **nextDigitalIO**( ) |

**Returns :**

a pointer to a `YDigitalIO` object, corresponding to a digital IO port currently online, or a `null` pointer if there are no more digital IO ports to enumerate.

## digitalio→pulse()[digitalio pulse: ] YDigitalIO

Triggers a pulse on a single bit for a specified duration.

| | |
|---|---|
| js | function **pulse**( **bitno**, **ms_duration**) |
| nodejs | function **pulse**( **bitno**, **ms_duration**) |
| php | function **pulse**( **$bitno**, **$ms_duration**) |
| cpp | int **pulse**( int **bitno**, int **ms_duration**) |
| m | -(int) **pulse** : (int) **bitno** : (int) **ms_duration** |
| pas | function **pulse**( **bitno**: LongInt, **ms_duration**: LongInt): LongInt |
| vb | function **pulse**( ) As Integer |
| cs | int **pulse**( int **bitno**, int **ms_duration**) |
| java | int **pulse**( int **bitno**, int **ms_duration**) |
| py | def **pulse**( **bitno**, **ms_duration**) |
| cmd | YDigitalIO **target** **pulse** **bitno** **ms_duration** |

The specified bit will be turned to 1, and then back to 0 after the given duration.

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**ms_duration** desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→registerValueCallback()[digitalio registerValueCallback: ] YDigitalIO

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YDigitalIOValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YDigitalIOValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYDigitalIOValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## digitalio→set_bitDirection() digitalio→setBitDirection()[digitalio setBitDirection: ] YDigitalIO

Changes the direction of a single bit from the I/O port.

```
js      function set_bitDirection( bitno, bitdirection)
nodejs  function set_bitDirection( bitno, bitdirection)
php     function set_bitDirection( $bitno, $bitdirection)
cpp     int set_bitDirection( int bitno, int bitdirection)
m       -(int) setBitDirection : (int) bitno : (int) bitdirection
pas     function set_bitDirection( bitno: LongInt, bitdirection: LongInt): LongInt
vb      function set_bitDirection( ) As Integer
cs      int set_bitDirection( int bitno, int bitdirection)
java    int set_bitDirection( int bitno, int bitdirection)
py      def set_bitDirection( bitno, bitdirection)
cmd     YDigitalIO target set_bitDirection bitno bitdirection
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**bitdirection** direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_bitOpenDrain() digitalio→setBitOpenDrain()[digitalio setBitOpenDrain: ] YDigitalIO

Changes the electrical interface of a single bit from the I/O port.

| | |
|---|---|
| js | function **set_bitOpenDrain**( **bitno**, **opendrain**) |
| nodejs | function **set_bitOpenDrain**( **bitno**, **opendrain**) |
| php | function **set_bitOpenDrain**( **$bitno**, **$opendrain**) |
| cpp | int **set_bitOpenDrain**( int **bitno**, int **opendrain**) |
| m | -(int) setBitOpenDrain : (int) **bitno** : (int) **opendrain** |
| pas | function **set_bitOpenDrain**( **bitno**: LongInt, **opendrain**: LongInt): LongInt |
| vb | function **set_bitOpenDrain**( ) As Integer |
| cs | int **set_bitOpenDrain**( int **bitno**, int **opendrain**) |
| java | int **set_bitOpenDrain**( int **bitno**, int **opendrain**) |
| py | def **set_bitOpenDrain**( **bitno**, **opendrain**) |
| cmd | YDigitalIO **target set_bitOpenDrain bitno opendrain** |

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**opendrain** 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_bitPolarity() digitalio→setBitPolarity()[digitalio setBitPolarity: ] YDigitalIO

Changes the polarity of a single bit from the I/O port.

| | |
|---|---|
| js | function **set_bitPolarity**( **bitno**, **bitpolarity**) |
| nodejs | function **set_bitPolarity**( **bitno**, **bitpolarity**) |
| php | function **set_bitPolarity**( **$bitno**, **$bitpolarity**) |
| cpp | int **set_bitPolarity**( int **bitno**, int **bitpolarity**) |
| m | -(int) setBitPolarity : (int) **bitno** : (int) **bitpolarity** |
| pas | function **set_bitPolarity**( **bitno**: LongInt, **bitpolarity**: LongInt): LongInt |
| vb | function **set_bitPolarity**( ) As Integer |
| cs | int **set_bitPolarity**( int **bitno**, int **bitpolarity**) |
| java | int **set_bitPolarity**( int **bitno**, int **bitpolarity**) |
| py | def **set_bitPolarity**( **bitno**, **bitpolarity**) |
| cmd | YDigitalIO **target** **set_bitPolarity** **bitno** **bitpolarity** |

**Parameters :**

**bitno** the bit number; lowest bit has index 0.

**bitpolarity** polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_bitState() digitalio→setBitState()[digitalio setBitState: ] YDigitalIO

Sets a single bit of the I/O port.

| | |
|---|---|
| js | function **set_bitState**( **bitno**, **bitstate**) |
| nodejs | function **set_bitState**( **bitno**, **bitstate**) |
| php | function **set_bitState**( **$bitno**, **$bitstate**) |
| cpp | int **set_bitState**( int **bitno**, int **bitstate**) |
| m | -(int) setBitState : (int) **bitno** : (int) **bitstate** |
| pas | function **set_bitState**( **bitno**: LongInt, **bitstate**: LongInt): LongInt |
| vb | function **set_bitState**( ) As Integer |
| cs | int **set_bitState**( int **bitno**, int **bitstate**) |
| java | int **set_bitState**( int **bitno**, int **bitstate**) |
| py | def **set_bitState**( **bitno**, **bitstate**) |
| cmd | YDigitalIO **target** **set_bitState** **bitno** **bitstate** |

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**bitstate** the state of the bit (1 or 0)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_logicalName() digitalio→setLogicalName()[digitalio setLogicalName: ]

**YDigitalIO**

Changes the logical name of the digital IO port.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YDigitalIO **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the digital IO port.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## digitalio→set_outputVoltage() digitalio→setOutputVoltage()[digitalio setOutputVoltage: ] YDigitalIO

Changes the voltage source used to drive output bits.

| | |
|---|---|
| js | function **set_outputVoltage**( **newval**) |
| nodejs | function **set_outputVoltage**( **newval**) |
| php | function **set_outputVoltage**( **$newval**) |
| cpp | int **set_outputVoltage**( Y_OUTPUTVOLTAGE_enum **newval**) |
| m | -(int) setOutputVoltage : (Y_OUTPUTVOLTAGE_enum) **newval** |
| pas | function **set_outputVoltage**( **newval**: Integer): integer |
| vb | function **set_outputVoltage**( ByVal **newval** As Integer) As Integer |
| cs | int **set_outputVoltage**( int **newval**) |
| java | int **set_outputVoltage**( int **newval**) |
| py | def **set_outputVoltage**( **newval**) |
| cmd | YDigitalIO **target** **set_outputVoltage** **newval** |

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Parameters :**

**newval** a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_portDirection() digitalio→setPortDirection()[digitalio setPortDirection: ] YDigitalIO

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

| | |
|---|---|
| js | function **set_portDirection**( **newval**) |
| nodejs | function **set_portDirection**( **newval**) |
| php | function **set_portDirection**( **$newval**) |
| cpp | int **set_portDirection**( int **newval**) |
| m | -(int) setPortDirection : (int) **newval** |
| pas | function **set_portDirection**( **newval**: LongInt): integer |
| vb | function **set_portDirection**( ByVal **newval** As Integer) As Integer |
| cs | int **set_portDirection**( int **newval**) |
| java | int **set_portDirection**( int **newval**) |
| py | def **set_portDirection**( **newval**) |
| cmd | YDigitalIO **target** **set_portDirection** **newval** |

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Parameters :**

**newval** an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_portOpenDrain() digitalio→setPortOpenDrain()[digitalio setPortOpenDrain: ] YDigitalIO

Changes the electrical interface for each bit of the port.

| | |
|---|---|
| js | function **set_portOpenDrain**( **newval**) |
| nodejs | function **set_portOpenDrain**( **newval**) |
| php | function **set_portOpenDrain**( **$newval**) |
| cpp | int **set_portOpenDrain**( int **newval**) |
| m | -(int) setPortOpenDrain : (int) **newval** |
| pas | function **set_portOpenDrain**( **newval**: LongInt): integer |
| vb | function **set_portOpenDrain**( ByVal **newval** As Integer) As Integer |
| cs | int **set_portOpenDrain**( int **newval**) |
| java | int **set_portOpenDrain**( int **newval**) |
| py | def **set_portOpenDrain**( **newval**) |
| cmd | YDigitalIO **target** **set_portOpenDrain** **newval** |

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Parameters :**

**newval** an integer corresponding to the electrical interface for each bit of the port

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_portPolarity() digitalio→setPortPolarity()[digitalio setPortPolarity: ] YDigitalIO

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

| | |
|---|---|
| js | function **set_portPolarity**( **newval**) |
| nodejs | function **set_portPolarity**( **newval**) |
| php | function **set_portPolarity**( **$newval**) |
| cpp | int **set_portPolarity**( int **newval**) |
| m | -(int) setPortPolarity : (int) **newval** |
| pas | function **set_portPolarity**( **newval**: LongInt): integer |
| vb | function **set_portPolarity**( ByVal **newval** As Integer) As Integer |
| cs | int **set_portPolarity**( int **newval**) |
| java | int **set_portPolarity**( int **newval**) |
| py | def **set_portPolarity**( **newval**) |
| cmd | YDigitalIO **target** **set_portPolarity** **newval** |

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

**Parameters :**

**newval** an integer corresponding to the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_portState() digitalio→setPortState()[digitalio setPortState: ] YDigitalIO

Changes the digital IO port state: bit 0 represents input 0, and so on.

| | |
|---|---|
| js | function **set_portState**( **newval**) |
| nodejs | function **set_portState**( **newval**) |
| php | function **set_portState**( **$newval**) |
| cpp | int **set_portState**( int **newval**) |
| m | -(int) setPortState : (int) **newval** |
| pas | function **set_portState**( **newval**: LongInt): integer |
| vb | function **set_portState**( ByVal **newval** As Integer) As Integer |
| cs | int **set_portState**( int **newval**) |
| java | int **set_portState**( int **newval**) |
| py | def **set_portState**( **newval**) |
| cmd | YDigitalIO **target** **set_portState** **newval** |

This function has no effect on bits configured as input in `portDirection`.

**Parameters :**

**newval** an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→set_userData() digitalio→setUserData()[digitalio setUserData: ] YDigitalIO

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## digitalio→toggle_bitState()[digitalio toggle_bitState: ] YDigitalIO

Reverts a single bit of the I/O port.

| | |
|---|---|
| js | function **toggle_bitState**( **bitno**) |
| nodejs | function **toggle_bitState**( **bitno**) |
| php | function **toggle_bitState**( **$bitno**) |
| cpp | int **toggle_bitState**( int **bitno**) |
| m | -(int) **toggle_bitState** : (int) **bitno** |
| pas | function **toggle_bitState**( **bitno**: LongInt): LongInt |
| vb | function **toggle_bitState**( ) As Integer |
| cs | int **toggle_bitState**( int **bitno**) |
| java | int **toggle_bitState**( int **bitno**) |
| py | def **toggle_bitState**( **bitno**) |
| cmd | YDigitalIO **target** **toggle_bitState** **bitno** |

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## digitalio→wait_async() YDigitalIO

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.13. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_display.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YDisplay = yoctolib.YDisplay; |
| php | require_once('yocto_display.php'); |
| cpp | #include "yocto_display.h" |
| m | #import "yocto_display.h" |
| pas | uses yocto_display; |
| vb | yocto_display.vb |
| cs | yocto_display.cs |
| java | import com.yoctopuce.YoctoAPI.YDisplay; |
| py | from yocto_display import * |

### Global functions

**yFindDisplay(func)**

Retrieves a display for a given identifier.

**yFirstDisplay()**

Starts the enumeration of displays currently accessible.

### `YDisplay` methods

**display→copyLayerContent(srcLayerId, dstLayerId)**

Copies the whole content of a layer to another layer.

**display→describe()**

Returns a short text that describes unambiguously the instance of the display in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**display→fade(brightness, duration)**

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

**display→get_advertisedValue()**

Returns the current value of the display (no more than 6 characters).

**display→get_brightness()**

Returns the luminosity of the module informative leds (from 0 to 100).

**display→get_displayHeight()**

Returns the display height, in pixels.

**display→get_displayLayer(layerId)**

Returns a YDisplayLayer object that can be used to draw on the specified layer.

**display→get_displayType()**

Returns the display type: monochrome, gray levels or full color.

**display→get_displayWidth()**

Returns the display width, in pixels.

**display→get_enabled()**

Returns true if the screen is powered, false otherwise.

**display→get_errorMessage()**

Returns the error message of the latest error with the display.

**display→get_errorType()**
Returns the numerical error code of the latest error with the display.

**display→get_friendlyName()**
Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

**display→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**display→get_functionId()**
Returns the hardware identifier of the display, without reference to the module.

**display→get_hardwareId()**
Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

**display→get_layerCount()**
Returns the number of available layers to draw on.

**display→get_layerHeight()**
Returns the height of the layers to draw on, in pixels.

**display→get_layerWidth()**
Returns the width of the layers to draw on, in pixels.

**display→get_logicalName()**
Returns the logical name of the display.

**display→get_module()**
Gets the `YModule` object for the device on which the function is located.

**display→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**display→get_orientation()**
Returns the currently selected display orientation.

**display→get_startupSeq()**
Returns the name of the sequence to play when the displayed is powered on.

**display→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**display→isOnline()**
Checks if the display is currently reachable, without raising any error.

**display→isOnline_async(callback, context)**
Checks if the display is currently reachable, without raising any error (asynchronous version).

**display→load(msValidity)**
Preloads the display cache with a specified validity duration.

**display→load_async(msValidity, callback, context)**
Preloads the display cache with a specified validity duration (asynchronous version).

**display→newSequence()**
Starts to record all display commands into a sequence, for later replay.

**display→nextDisplay()**
Continues the enumeration of displays started using `yFirstDisplay()`.

**display→pauseSequence(delay_ms)**
Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

**display→playSequence(sequenceName)**
Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

**display→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**display→resetAll()**

Clears the display screen and resets all display layers to their default state.

**display→saveSequence(sequenceName)**

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

**display→set_brightness(newval)**

Changes the brightness of the display.

**display→set_enabled(newval)**

Changes the power state of the display.

**display→set_logicalName(newval)**

Changes the logical name of the display.

**display→set_orientation(newval)**

Changes the display orientation.

**display→set_startupSeq(newval)**

Changes the name of the sequence to play when the displayed is powered on.

**display→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**display→stopSequence()**

Stops immediately any ongoing sequence replay.

**display→swapLayerContent(layerIdA, layerIdB)**

Swaps the whole content of two layers.

**display→upload(pathname, content)**

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

**display→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDisplay.FindDisplay() yFindDisplay()yFindDisplay() — YDisplay

Retrieves a display for a given identifier.

| | |
|---|---|
| js | function **yFindDisplay**( **func**) |
| nodejs | function **FindDisplay**( **func**) |
| php | function **yFindDisplay**( **$func**) |
| cpp | YDisplay* **yFindDisplay**( string **func**) |
| m | +(YDisplay*) **yFindDisplay** : (NSString*) **func** |
| pas | function **yFindDisplay**( **func**: string): TYDisplay |
| vb | function **yFindDisplay**( ByVal **func** As String) As YDisplay |
| cs | YDisplay **FindDisplay**( string **func**) |
| java | YDisplay **FindDisplay**( String **func**) |
| py | def **FindDisplay**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the display

**Returns :**

a `YDisplay` object allowing you to drive the display.

## YDisplay.FirstDisplay() yFirstDisplay()yFirstDisplay() YDisplay

Starts the enumeration of displays currently accessible.

| | |
|---|---|
| js | function **yFirstDisplay**( ) |
| nodejs | function **FirstDisplay**( ) |
| php | function **yFirstDisplay**( ) |
| cpp | YDisplay* **yFirstDisplay**( ) |
| m | YDisplay* **yFirstDisplay**( ) |
| pas | function **yFirstDisplay**( ): TYDisplay |
| vb | function **yFirstDisplay**( ) As YDisplay |
| cs | YDisplay **FirstDisplay**( ) |
| java | YDisplay **FirstDisplay**( ) |
| py | def **FirstDisplay**( ) |

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

**Returns :**

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

## display→copyLayerContent()[display copyLayerContent: ] YDisplay

Copies the whole content of a layer to another layer.

| | |
|---|---|
| js | function **copyLayerContent**( **srcLayerId**, **dstLayerId**) |
| nodejs | function **copyLayerContent**( **srcLayerId**, **dstLayerId**) |
| php | function **copyLayerContent**( **$srcLayerId**, **$dstLayerId**) |
| cpp | int **copyLayerContent**( int **srcLayerId**, int **dstLayerId**) |
| m | -(int) **copyLayerContent** : (int) **srcLayerId**<br>: (int) **dstLayerId** |
| pas | function **copyLayerContent**( **srcLayerId**: LongInt,<br>**dstLayerId**: LongInt): LongInt |
| vb | function **copyLayerContent**( ) As Integer |
| cs | int **copyLayerContent**( int **srcLayerId**, int **dstLayerId**) |
| java | int **copyLayerContent**( int **srcLayerId**, int **dstLayerId**) |
| py | def **copyLayerContent**( **srcLayerId**, **dstLayerId**) |
| cmd | YDisplay **target 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`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the 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.

| | |
|---|---|
| js | function **fade**( **brightness**, **duration**) |
| nodejs | function **fade**( **brightness**, **duration**) |
| php | function **fade**( **$brightness**, **$duration**) |
| cpp | int **fade**( int **brightness**, int **duration**) |
| m | -(int) **fade** : (int) **brightness** : (int) **duration** |
| pas | function **fade**( **brightness**: LongInt, **duration**: LongInt): LongInt |
| vb | function **fade**( ) As Integer |
| cs | int **fade**( int **brightness**, int **duration**) |
| java | int **fade**( int **brightness**, int **duration**) |
| py | def **fade**( **brightness**, **duration**) |
| cmd | YDisplay **target** **fade** **brightness** **duration** |

**Parameters :**

**brightness** the new screen brightness

**duration** duration of the brightness transition, in milliseconds.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→get_advertisedValue() display→advertisedValue()[display advertisedValue] YDisplay

Returns the current value of the display (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YDisplay **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the display (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## display→get_brightness() display→brightness()[display brightness] YDisplay

Returns the luminosity of the module informative leds (from 0 to 100).

| | |
|---|---|
| js | function **get_brightness**( ) |
| nodejs | function **get_brightness**( ) |
| php | function **get_brightness**( ) |
| cpp | int **get_brightness**( ) |
| m | -(int) brightness |
| pas | function **get_brightness**( ): LongInt |
| vb | function **get_brightness**( ) As Integer |
| cs | int **get_brightness**( ) |
| java | int **get_brightness**( ) |
| py | def **get_brightness**( ) |
| cmd | YDisplay **target** **get_brightness** |

**Returns :**
an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_BRIGHTNESS_INVALID`.

## display→get_displayHeight() display→displayHeight()[display displayHeight] YDisplay

Returns the display height, in pixels.

| | |
|---|---|
| js | function **get_displayHeight**( ) |
| nodejs | function **get_displayHeight**( ) |
| php | function **get_displayHeight**( ) |
| cpp | int **get_displayHeight**( ) |
| m | -(int) displayHeight |
| pas | function **get_displayHeight**( ): LongInt |
| vb | function **get_displayHeight**( ) As Integer |
| cs | int **get_displayHeight**( ) |
| java | int **get_displayHeight**( ) |
| py | def **get_displayHeight**( ) |
| cmd | YDisplay **target** **get_displayHeight** |

**Returns :**
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns `Y_DISPLAYHEIGHT_INVALID`.

## display→get_displayLayer() display→displayLayer()[display displayLayer: ] YDisplay

Returns a YDisplayLayer object that can be used to draw on the specified layer.

| | |
|---|---|
| js | function **get_displayLayer**( **layerId**) |
| nodejs | function **get_displayLayer**( **layerId**) |
| php | function **get_displayLayer**( **$layerId**) |
| cpp | YDisplayLayer* **get_displayLayer**( unsigned **layerId**) |
| m | -(YDisplayLayer*) displayLayer : (unsigned) **layerId** |
| vb | function **get_displayLayer**( ) As YDisplayLayer |
| cs | YDisplayLayer **get_displayLayer**( int **layerId**) |
| java | synchronized YDisplayLayer **get_displayLayer**( int **layerId**) |
| py | def **get_displayLayer**( **layerId**) |

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

**Parameters :**

**layerId** the identifier of the layer (a number in range 0..layerCount-1)

**Returns :**

an `YDisplayLayer` object

On failure, throws an exception or returns `null`.

## display→get_displayType() display→displayType()[display displayType] YDisplay

Returns the display type: monochrome, gray levels or full color.

| | |
|---|---|
| js | function **get_displayType**( ) |
| nodejs | function **get_displayType**( ) |
| php | function **get_displayType**( ) |
| cpp | Y_DISPLAYTYPE_enum **get_displayType**( ) |
| m | -(Y_DISPLAYTYPE_enum) displayType |
| pas | function **get_displayType**( ): Integer |
| vb | function **get_displayType**( ) As Integer |
| cs | int **get_displayType**( ) |
| java | int **get_displayType**( ) |
| py | def **get_displayType**( ) |
| cmd | YDisplay **target** **get_displayType** |

**Returns :**

a value among `Y_DISPLAYTYPE_MONO`, `Y_DISPLAYTYPE_GRAY` and `Y_DISPLAYTYPE_RGB` corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns `Y_DISPLAYTYPE_INVALID`.

## display→get_displayWidth() display→displayWidth()[display displayWidth] YDisplay

Returns the display width, in pixels.

| | |
|---|---|
| js | function **get_displayWidth**( ) |
| nodejs | function **get_displayWidth**( ) |
| php | function **get_displayWidth**( ) |
| cpp | int **get_displayWidth**( ) |
| m | -(int) displayWidth |
| pas | function **get_displayWidth**( ): LongInt |
| vb | function **get_displayWidth**( ) As Integer |
| cs | int **get_displayWidth**( ) |
| java | int **get_displayWidth**( ) |
| py | def **get_displayWidth**( ) |
| cmd | YDisplay **target** **get_displayWidth** |

**Returns :**
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns `Y_DISPLAYWIDTH_INVALID`.

## display→get_enabled() display→enabled()[display enabled] — YDisplay

Returns true if the screen is powered, false otherwise.

| | |
|---|---|
| js | function **get_enabled**( ) |
| nodejs | function **get_enabled**( ) |
| php | function **get_enabled**( ) |
| cpp | Y_ENABLED_enum **get_enabled**( ) |
| m | -(Y_ENABLED_enum) enabled |
| pas | function **get_enabled**( ): Integer |
| vb | function **get_enabled**( ) As Integer |
| cs | int **get_enabled**( ) |
| java | int **get_enabled**( ) |
| py | def **get_enabled**( ) |
| cmd | YDisplay **target** **get_enabled** |

**Returns :**

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

## display→get_errorMessage() display→errorMessage()[display errorMessage] YDisplay

Returns the error message of the latest error with the display.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the display object

## display→get_errorType() display→errorType() YDisplay

Returns the numerical error code of the latest error with the display.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the display object

## display→get_friendlyName() display→friendlyName()[display friendlyName] YDisplay

Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the display if they are defined, otherwise the serial number of the module and the hardware identifier of the display (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the display using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## display→get_functionDescriptor() display→functionDescriptor()[display functionDescriptor] YDisplay

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## display→get_functionId() display→functionId()[display functionId] YDisplay

Returns the hardware identifier of the display, without reference to the module.

| | |
|---|---|
| `js` | function **get_functionId**( ) |
| `nodejs` | function **get_functionId**( ) |
| `php` | function **get_functionId**( ) |
| `cpp` | string **get_functionId**( ) |
| `m` | -(NSString*) functionId |
| `vb` | function **get_functionId**( ) As String |
| `cs` | string **get_functionId**( ) |
| `java` | String **get_functionId**( ) |
| `py` | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the display (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## display→get_hardwareId() display→hardwareId()[display hardwareId] YDisplay

Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the display (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## display→get_layerCount() display→layerCount()[display layerCount] YDisplay

Returns the number of available layers to draw on.

| | |
|---|---|
| js | function **get_layerCount**( ) |
| nodejs | function **get_layerCount**( ) |
| php | function **get_layerCount**( ) |
| cpp | int **get_layerCount**( ) |
| m | -(int) layerCount |
| pas | function **get_layerCount**( ): LongInt |
| vb | function **get_layerCount**( ) As Integer |
| cs | int **get_layerCount**( ) |
| java | int **get_layerCount**( ) |
| py | def **get_layerCount**( ) |
| cmd | YDisplay **target** **get_layerCount** |

**Returns :**
an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns `Y_LAYERCOUNT_INVALID`.

## display→get_layerHeight() display→layerHeight()[display layerHeight] YDisplay

Returns the height of the layers to draw on, in pixels.

| | |
|---|---|
| js | function **get_layerHeight**( ) |
| nodejs | function **get_layerHeight**( ) |
| php | function **get_layerHeight**( ) |
| cpp | int **get_layerHeight**( ) |
| m | -(int) layerHeight |
| pas | function **get_layerHeight**( ): LongInt |
| vb | function **get_layerHeight**( ) As Integer |
| cs | int **get_layerHeight**( ) |
| java | int **get_layerHeight**( ) |
| py | def **get_layerHeight**( ) |
| cmd | YDisplay **target** **get_layerHeight** |

**Returns :**
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns `Y_LAYERHEIGHT_INVALID`.

## display→get_layerWidth() display→layerWidth()[display layerWidth]

**YDisplay**

Returns the width of the layers to draw on, in pixels.

| | |
|---|---|
| js | function **get_layerWidth**( ) |
| nodejs | function **get_layerWidth**( ) |
| php | function **get_layerWidth**( ) |
| cpp | int **get_layerWidth**( ) |
| m | -(int) layerWidth |
| pas | function **get_layerWidth**( ): LongInt |
| vb | function **get_layerWidth**( ) As Integer |
| cs | int **get_layerWidth**( ) |
| java | int **get_layerWidth**( ) |
| py | def **get_layerWidth**( ) |
| cmd | YDisplay **target** **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 logicalName]

Returns the logical name of the display.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YDisplay **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the display. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## display→get_module() display→module()[display module] YDisplay

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## display→get_module_async() display→module_async() **YDisplay**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## display→get_orientation() display→orientation()[display orientation]   YDisplay

Returns the currently selected display orientation.

| | |
|---|---|
| js | function **get_orientation**( ) |
| nodejs | function **get_orientation**( ) |
| php | function **get_orientation**( ) |
| cpp | Y_ORIENTATION_enum **get_orientation**( ) |
| m | -(Y_ORIENTATION_enum) orientation |
| pas | function **get_orientation**( ): Integer |
| vb | function **get_orientation**( ) As Integer |
| cs | int **get_orientation**( ) |
| java | int **get_orientation**( ) |
| py | def **get_orientation**( ) |
| cmd | YDisplay **target** **get_orientation** |

**Returns :**

a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the currently selected display orientation

On failure, throws an exception or returns `Y_ORIENTATION_INVALID`.

## display→get_startupSeq() display→startupSeq()[display startupSeq] YDisplay

Returns the name of the sequence to play when the displayed is powered on.

| | |
|---|---|
| js | function **get_startupSeq**( ) |
| nodejs | function **get_startupSeq**( ) |
| php | function **get_startupSeq**( ) |
| cpp | string **get_startupSeq**( ) |
| m | -(NSString*) startupSeq |
| pas | function **get_startupSeq**( ): string |
| vb | function **get_startupSeq**( ) As String |
| cs | string **get_startupSeq**( ) |
| java | String **get_startupSeq**( ) |
| py | def **get_startupSeq**( ) |
| cmd | YDisplay **target** **get_startupSeq** |

**Returns :**
a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns `Y_STARTUPSEQ_INVALID`.

## display→get_userData() display→userData()[display userData]

YDisplay

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## display→isOnline()[display isOnline] YDisplay

Checks if the display is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

**Returns :**
`true` if the display can be reached, and `false` otherwise

## display→isOnline_async() YDisplay

Checks if the display is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## display→load()[display load: ] YDisplay

Preloads the display cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## display→load_async() YDisplay

Preloads the display cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## display→newSequence()[display newSequence] YDisplay

Starts to record all display commands into a sequence, for later replay.

| | |
|---|---|
| js | function **newSequence**( ) |
| nodejs | function **newSequence**( ) |
| php | function **newSequence**( ) |
| cpp | int **newSequence**( ) |
| m | -(int) **newSequence** |
| pas | function **newSequence**( ): LongInt |
| vb | function **newSequence**( ) As Integer |
| cs | int **newSequence**( ) |
| java | int **newSequence**( ) |
| py | def **newSequence**( ) |
| cmd | YDisplay **target** **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()`.

| | |
|---|---|
| js | function **nextDisplay**( ) |
| nodejs | function **nextDisplay**( ) |
| php | function **nextDisplay**( ) |
| cpp | YDisplay * **nextDisplay**( ) |
| m | -(YDisplay*) **nextDisplay** |
| pas | function **nextDisplay**( ): TYDisplay |
| vb | function **nextDisplay**( ) As YDisplay |
| cs | YDisplay **nextDisplay**( ) |
| java | YDisplay **nextDisplay**( ) |
| py | def **nextDisplay**( ) |

**Returns :**

a pointer to a `YDisplay` object, corresponding to a display currently online, or a `null` pointer if there are no more displays to enumerate.

## display→pauseSequence()[display pauseSequence: ] YDisplay

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

| | |
|---|---|
| js | function **pauseSequence**( **delay_ms**) |
| nodejs | function **pauseSequence**( **delay_ms**) |
| php | function **pauseSequence**( **$delay_ms**) |
| cpp | int **pauseSequence**( int **delay_ms**) |
| m | -(int) **pauseSequence** : (int) **delay_ms** |
| pas | function **pauseSequence**( **delay_ms**: LongInt): LongInt |
| vb | function **pauseSequence**( ) As Integer |
| cs | int **pauseSequence**( int **delay_ms**) |
| java | int **pauseSequence**( int **delay_ms**) |
| py | def **pauseSequence**( **delay_ms**) |
| cmd | YDisplay **target** **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()`.

| | |
|---|---|
| js | function **playSequence**( **sequenceName**) |
| nodejs | function **playSequence**( **sequenceName**) |
| php | function **playSequence**( **$sequenceName**) |
| cpp | int **playSequence**( string **sequenceName**) |
| m | -(int) **playSequence** : (NSString*) **sequenceName** |
| pas | function **playSequence**( **sequenceName**: string): LongInt |
| vb | function **playSequence**( ) As Integer |
| cs | int **playSequence**( string **sequenceName**) |
| java | int **playSequence**( String **sequenceName**) |
| py | def **playSequence**( **sequenceName**) |
| cmd | YDisplay **target** **playSequence** **sequenceName** |

**Parameters :**

**sequenceName** the name of the newly created sequence

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→registerValueCallback()[display registerValueCallback: ] YDisplay

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YDisplayValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YDisplayValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYDisplayValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## display→resetAll()[display resetAll] YDisplay

Clears the display screen and resets all display layers to their default state.

| | |
|---|---|
| js | function **resetAll**( ) |
| nodejs | function **resetAll**( ) |
| php | function **resetAll**( ) |
| cpp | int **resetAll**( ) |
| m | -(int) **resetAll** |
| pas | function **resetAll**( ): LongInt |
| vb | function **resetAll**( ) As Integer |
| cs | int **resetAll**( ) |
| java | int **resetAll**( ) |
| py | def **resetAll**( ) |
| cmd | YDisplay **target** **resetAll** |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→saveSequence()[display saveSequence: ] YDisplay

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

| | |
|---|---|
| js | function **saveSequence**( **sequenceName**) |
| nodejs | function **saveSequence**( **sequenceName**) |
| php | function **saveSequence**( **$sequenceName**) |
| cpp | int **saveSequence**( string **sequenceName**) |
| m | -(int) **saveSequence** : (NSString*) **sequenceName** |
| pas | function **saveSequence**( **sequenceName**: string): LongInt |
| vb | function **saveSequence**( ) As Integer |
| cs | int **saveSequence**( string **sequenceName**) |
| java | int **saveSequence**( String **sequenceName**) |
| py | def **saveSequence**( **sequenceName**) |
| cmd | YDisplay **target** **saveSequence** **sequenceName** |

The sequence can be later replayed using `playSequence()`.

**Parameters :**

**sequenceName** the name of the newly created sequence

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→set_brightness() display→setBrightness()[display setBrightness: ] YDisplay

Changes the brightness of the display.

| | |
|---|---|
| js | function **set_brightness**( **newval**) |
| nodejs | function **set_brightness**( **newval**) |
| php | function **set_brightness**( **$newval**) |
| cpp | int **set_brightness**( int **newval**) |
| m | -(int) setBrightness : (int) **newval** |
| pas | function **set_brightness**( **newval**: LongInt): integer |
| vb | function **set_brightness**( ByVal **newval** As Integer) As Integer |
| cs | int **set_brightness**( int **newval**) |
| java | int **set_brightness**( int **newval**) |
| py | def **set_brightness**( **newval**) |
| cmd | YDisplay **target set_brightness newval** |

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the brightness of the display

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→set_enabled() display→setEnabled()[display setEnabled: ] — YDisplay

Changes the power state of the display.

| | |
|---|---|
| js | function **set_enabled**( **newval**) |
| nodejs | function **set_enabled**( **newval**) |
| php | function **set_enabled**( **$newval**) |
| cpp | int **set_enabled**( Y_ENABLED_enum **newval**) |
| m | -(int) setEnabled : (Y_ENABLED_enum) **newval** |
| pas | function **set_enabled**( **newval**: Integer): integer |
| vb | function **set_enabled**( ByVal **newval** As Integer) As Integer |
| cs | int **set_enabled**( int **newval**) |
| java | int **set_enabled**( int **newval**) |
| py | def **set_enabled**( **newval**) |
| cmd | YDisplay **target** **set_enabled** **newval** |

**Parameters :**

**newval** either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the power state of the display

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→set_logicalName() display→setLogicalName()[display setLogicalName: ] YDisplay

Changes the logical name of the display.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YDisplay **target set_logicalName newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the display.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## display→set_orientation() display→setOrientation()[display setOrientation: ] YDisplay

Changes the display orientation.

| | |
|---|---|
| js | function **set_orientation**( **newval**) |
| nodejs | function **set_orientation**( **newval**) |
| php | function **set_orientation**( **$newval**) |
| cpp | int **set_orientation**( Y_ORIENTATION_enum **newval**) |
| m | -(int) setOrientation : (Y_ORIENTATION_enum) **newval** |
| pas | function **set_orientation**( **newval**: Integer): integer |
| vb | function **set_orientation**( ByVal **newval** As Integer) As Integer |
| cs | int **set_orientation**( int **newval**) |
| java | int **set_orientation**( int **newval**) |
| py | def **set_orientation**( **newval**) |
| cmd | YDisplay **target** **set_orientation** **newval** |

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the display orientation

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→set_startupSeq() display→setStartupSeq()[display setStartupSeq: ] YDisplay

Changes the name of the sequence to play when the displayed is powered on.

| | |
|---|---|
| js | function **set_startupSeq**( **newval**) |
| nodejs | function **set_startupSeq**( **newval**) |
| php | function **set_startupSeq**( **$newval**) |
| cpp | int **set_startupSeq**( const string& **newval**) |
| m | -(int) setStartupSeq : (NSString*) **newval** |
| pas | function **set_startupSeq**( **newval**: string): integer |
| vb | function **set_startupSeq**( ByVal **newval** As String) As Integer |
| cs | int **set_startupSeq**( string **newval**) |
| java | int **set_startupSeq**( String **newval**) |
| py | def **set_startupSeq**( **newval**) |
| cmd | YDisplay **target** **set_startupSeq** **newval** |

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the name of the sequence to play when the displayed is powered on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→set_userData() display→setUserData()[display setUserData: ] YDisplay

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## display→stopSequence()[display stopSequence] YDisplay

Stops immediately any ongoing sequence replay.

| | |
|---|---|
| js | function **stopSequence**( ) |
| nodejs | function **stopSequence**( ) |
| php | function **stopSequence**( ) |
| cpp | int **stopSequence**( ) |
| m | -(int) **stopSequence** |
| pas | function **stopSequence**( ): LongInt |
| vb | function **stopSequence**( ) As Integer |
| cs | int **stopSequence**( ) |
| java | int **stopSequence**( ) |
| py | def **stopSequence**( ) |
| cmd | YDisplay **target** **stopSequence** |

The display is left as is.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→swapLayerContent()[display swapLayerContent: ] YDisplay

Swaps the whole content of two layers.

| | |
|---|---|
| js | function **swapLayerContent**( **layerIdA**, **layerIdB**) |
| nodejs | function **swapLayerContent**( **layerIdA**, **layerIdB**) |
| php | function **swapLayerContent**( **$layerIdA**, **$layerIdB**) |
| cpp | int **swapLayerContent**( int **layerIdA**, int **layerIdB**) |
| m | -(int) **swapLayerContent** : (int) **layerIdA**<br>: (int) **layerIdB** |
| pas | function **swapLayerContent**( **layerIdA**: LongInt, **layerIdB**: LongInt): LongInt |
| vb | function **swapLayerContent**( ) As Integer |
| cs | int **swapLayerContent**( int **layerIdA**, int **layerIdB**) |
| java | int **swapLayerContent**( int **layerIdA**, int **layerIdB**) |
| py | def **swapLayerContent**( **layerIdA**, **layerIdB**) |
| cmd | YDisplay **target** **swapLayerContent** **layerIdA** **layerIdB** |

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between onae hidden layer and a visible layer, this method makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support (it is always completely opaque).

**Parameters :**

**layerIdA** the first layer (a number in range 0..layerCount-1)

**layerIdB** the second layer (a number in range 0..layerCount-1)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## display→upload()[display upload: ] YDisplay

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

| | |
|---|---|
| `js` | function **upload**( **pathname**, **content**) |
| `nodejs` | function **upload**( **pathname**, **content**) |
| `php` | function **upload**( **$pathname**, **$content**) |
| `cpp` | int **upload**( string **pathname**, string **content**) |
| `m` | -(int) **upload** : (NSString*) **pathname**<br>: (NSData*) **content** |
| `pas` | function **upload**( **pathname**: string, **content**: TByteArray): LongInt |
| `vb` | procedure **upload**( ) |
| `cs` | int **upload**( string **pathname**) |
| `java` | int **upload**( String **pathname**) |
| `py` | def **upload**( **pathname**, **content**) |
| `cmd` | YDisplay **target** **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() YDisplay

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.14. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_display.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YDisplay = yoctolib.YDisplay; |
| php | require_once('yocto_display.php'); |
| cpp | #include "yocto_display.h" |
| m | #import "yocto_display.h" |
| pas | uses yocto_display; |
| vb | yocto_display.vb |
| cs | yocto_display.cs |
| java | import com.yoctopuce.YoctoAPI.YDisplay; |
| py | from yocto_display import * |

| YDisplayLayer methods |
|---|
| **displaylayer→clear()**<br>Erases the whole content of the layer (makes it fully transparent). |
| **displaylayer→clearConsole()**<br>Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console. |
| **displaylayer→consoleOut(text)**<br>Outputs a message in the console area, and advances the console pointer accordingly. |
| **displaylayer→drawBar(x1, y1, x2, y2)**<br>Draws a filled rectangular bar at a specified position. |
| **displaylayer→drawBitmap(x, y, w, bitmap, bgcol)**<br>Draws a bitmap at the specified position. |
| **displaylayer→drawCircle(x, y, r)**<br>Draws an empty circle at a specified position. |
| **displaylayer→drawDisc(x, y, r)**<br>Draws a filled disc at a given position. |
| **displaylayer→drawImage(x, y, imagename)**<br>Draws a GIF image at the specified position. |
| **displaylayer→drawPixel(x, y)**<br>Draws a single pixel at the specified position. |
| **displaylayer→drawRect(x1, y1, x2, y2)**<br>Draws an empty rectangle at a specified position. |
| **displaylayer→drawText(x, y, anchor, text)**<br>Draws a text string at the specified position. |
| **displaylayer→get_display()**<br>Gets parent YDisplay. |
| **displaylayer→get_displayHeight()**<br>Returns the display height, in pixels. |
| **displaylayer→get_displayWidth()**<br>Returns the display width, in pixels. |

**displaylayer→get_layerHeight()**

Returns the height of the layers to draw on, in pixels.

**displaylayer→get_layerWidth()**

Returns the width of the layers to draw on, in pixels.

**displaylayer→hide()**

Hides the layer.

**displaylayer→lineTo(x, y)**

Draws a line from current drawing pointer position to the specified position.

**displaylayer→moveTo(x, y)**

Moves the drawing pointer of this layer to the specified position.

**displaylayer→reset()**

Reverts the layer to its initial state (fully transparent, default settings).

**displaylayer→selectColorPen(color)**

Selects the pen color for all subsequent drawing functions, including text drawing.

**displaylayer→selectEraser()**

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

**displaylayer→selectFont(fontname)**

Selects a font to use for the next text drawing functions, by providing the name of the font file.

**displaylayer→selectGrayPen(graylevel)**

Selects the pen gray level for all subsequent drawing functions, including text drawing.

**displaylayer→setAntialiasingMode(mode)**

Enables or disables anti-aliasing for drawing oblique lines and circles.

**displaylayer→setConsoleBackground(bgcol)**

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

**displaylayer→setConsoleMargins(x1, y1, x2, y2)**

Sets up display margins for the `consoleOut` function.

**displaylayer→setConsoleWordWrap(wordwrap)**

Sets up the wrapping behaviour used by the `consoleOut` function.

**displaylayer→setLayerPosition(x, y, scrollTime)**

Sets the position of the layer relative to the display upper left corner.

**displaylayer→unhide()**

Shows the layer.

## displaylayer→clear()[displaylayer clear] YDisplayLayer

Erases the whole content of the layer (makes it fully transparent).

| | |
|---|---|
| js | function **clear**( ) |
| nodejs | function **clear**( ) |
| php | function **clear**( ) |
| cpp | int **clear**( ) |
| m | -(int) **clear** |
| pas | function **clear**( ): LongInt |
| vb | function **clear**( ) As Integer |
| cs | int **clear**( ) |
| java | int **clear**( ) |
| py | def **clear**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **clear** |

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset()` instead.

**Returns :**
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→clearConsole()[displaylayer clearConsole] YDisplayLayer

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

| | |
|---|---|
| js | function **clearConsole**( ) |
| nodejs | function **clearConsole**( ) |
| php | function **clearConsole**( ) |
| cpp | int **clearConsole**( ) |
| m | -(int) **clearConsole** |
| pas | function **clearConsole**( ): LongInt |
| vb | function **clearConsole**( ) As Integer |
| cs | int **clearConsole**( ) |
| java | int **clearConsole**( ) |
| py | def **clearConsole**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| `js` | function **consoleOut**( **text**) |
| `nodejs` | function **consoleOut**( **text**) |
| `php` | function **consoleOut**( **$text**) |
| `cpp` | int **consoleOut**( string **text**) |
| `m` | -(int) **consoleOut** : (NSString*) **text** |
| `pas` | function **consoleOut**( **text**: string): LongInt |
| `vb` | function **consoleOut**( ) As Integer |
| `cs` | int **consoleOut**( string **text**) |
| `java` | int **consoleOut**( String **text**) |
| `py` | def **consoleOut**( **text**) |
| `cmd` | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawBar**( **x1**, **y1**, **x2**, **y2**) |
| nodejs | function **drawBar**( **x1**, **y1**, **x2**, **y2**) |
| php | function **drawBar**( **$x1**, **$y1**, **$x2**, **$y2**) |
| cpp | int **drawBar**( int **x1**, int **y1**, int **x2**, int **y2**) |
| m | -(int) **drawBar** : (int) **x1**<br>: (int) **y1**<br>: (int) **x2**<br>: (int) **y2** |
| pas | function **drawBar**( **x1**: LongInt,<br>**y1**: LongInt,<br>**x2**: LongInt,<br>**y2**: LongInt): LongInt |
| vb | function **drawBar**( ) As Integer |
| cs | int **drawBar**( int **x1**, int **y1**, int **x2**, int **y2**) |
| java | int **drawBar**( int **x1**, int **y1**, int **x2**, int **y2**) |
| py | def **drawBar**( **x1**, **y1**, **x2**, **y2**) |
| cmd | YDisplay **target** [-layer **layerId**] **drawBar** **x1** **y1** **x2** **y2** |

**Parameters :**

- **x1** the distance from left of layer to the left border of the rectangle, in pixels
- **y1** the distance from top of layer to the top border of the rectangle, in pixels
- **x2** the distance from left of layer to the right border of the rectangle, in pixels
- **y2** the distance from top of layer to the bottom border of the rectangle, in pixels

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→drawBitmap()[displaylayer drawBitmap: ] YDisplayLayer

Draws a bitmap at the specified position.

| | |
|---|---|
| js | function **drawBitmap**( **x**, **y**, **w**, **bitmap**, **bgcol**) |
| nodejs | function **drawBitmap**( **x**, **y**, **w**, **bitmap**, **bgcol**) |
| php | function **drawBitmap**( **$x**, **$y**, **$w**, **$bitmap**, **$bgcol**) |
| cpp | int **drawBitmap**( int **x**, int **y**, int **w**, string **bitmap**, int **bgcol**) |
| m | -(int) **drawBitmap** : (int) **x** : (int) **y** : (int) **w** : (NSData*) **bitmap** : (int) **bgcol** |
| pas | function **drawBitmap**( **x**: LongInt, **y**: LongInt, **w**: LongInt, **bitmap**: TByteArray, **bgcol**: LongInt): LongInt |
| vb | procedure **drawBitmap**( ) |
| cs | int **drawBitmap**( int **x**, int **y**, int **w**, int **bgcol**) |
| java | int **drawBitmap**( int **x**, int **y**, int **w**, int **bgcol**) |
| py | def **drawBitmap**( **x**, **y**, **w**, **bitmap**, **bgcol**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawCircle**( **x**, **y**, **r**) |
| nodejs | function **drawCircle**( **x**, **y**, **r**) |
| php | function **drawCircle**( **$x**, **$y**, **$r**) |
| cpp | int **drawCircle**( int **x**, int **y**, int **r**) |
| m | -(int) **drawCircle** : (int) **x** : (int) **y** : (int) **r** |
| pas | function **drawCircle**( **x**: LongInt, **y**: LongInt, **r**: LongInt): LongInt |
| vb | function **drawCircle**( ) As Integer |
| cs | int **drawCircle**( int **x**, int **y**, int **r**) |
| java | int **drawCircle**( int **x**, int **y**, int **r**) |
| py | def **drawCircle**( **x**, **y**, **r**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawDisc**( **x**, **y**, **r**) |
| nodejs | function **drawDisc**( **x**, **y**, **r**) |
| php | function **drawDisc**( **$x**, **$y**, **$r**) |
| cpp | int **drawDisc**( int **x**, int **y**, int **r**) |
| m | -(int) **drawDisc** : (int) **x**<br>: (int) **y**<br>: (int) **r** |
| pas | function **drawDisc**( **x**: LongInt, **y**: LongInt, **r**: LongInt): LongInt |
| vb | function **drawDisc**( ) As Integer |
| cs | int **drawDisc**( int **x**, int **y**, int **r**) |
| java | int **drawDisc**( int **x**, int **y**, int **r**) |
| py | def **drawDisc**( **x**, **y**, **r**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawImage**( **x**, **y**, **imagename**) |
| nodejs | function **drawImage**( **x**, **y**, **imagename**) |
| php | function **drawImage**( **$x**, **$y**, **$imagename**) |
| cpp | int **drawImage**( int **x**, int **y**, string **imagename**) |
| m | -(int) **drawImage** : (int) **x**<br>: (int) **y**<br>: (NSString*) **imagename** |
| pas | function **drawImage**( **x**: LongInt, **y**: LongInt, **imagename**: string): LongInt |
| vb | function **drawImage**( ) As Integer |
| cs | int **drawImage**( int **x**, int **y**, string **imagename**) |
| java | int **drawImage**( int **x**, int **y**, String **imagename**) |
| py | def **drawImage**( **x**, **y**, **imagename**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawPixel**( **x**, **y**) |
| nodejs | function **drawPixel**( **x**, **y**) |
| php | function **drawPixel**( **$x**, **$y**) |
| cpp | int **drawPixel**( int **x**, int **y**) |
| m | -(int) **drawPixel** : (int) **x**<br>: (int) **y** |
| pas | function **drawPixel**( **x**: LongInt, **y**: LongInt): LongInt |
| vb | function **drawPixel**( ) As Integer |
| cs | int **drawPixel**( int **x**, int **y**) |
| java | int **drawPixel**( int **x**, int **y**) |
| py | def **drawPixel**( **x**, **y**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawRect**( **x1**, **y1**, **x2**, **y2**) |
| nodejs | function **drawRect**( **x1**, **y1**, **x2**, **y2**) |
| php | function **drawRect**( **$x1**, **$y1**, **$x2**, **$y2**) |
| cpp | int **drawRect**( int **x1**, int **y1**, int **x2**, int **y2**) |
| m | -(int) **drawRect** : (int) **x1** : (int) **y1** : (int) **x2** : (int) **y2** |
| pas | function **drawRect**( **x1**: LongInt, **y1**: LongInt, **x2**: LongInt, **y2**: LongInt): LongInt |
| vb | function **drawRect**( ) As Integer |
| cs | int **drawRect**( int **x1**, int **y1**, int **x2**, int **y2**) |
| java | int **drawRect**( int **x1**, int **y1**, int **x2**, int **y2**) |
| py | def **drawRect**( **x1**, **y1**, **x2**, **y2**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **drawText**( **x**, **y**, **anchor**, **text**) |
| nodejs | function **drawText**( **x**, **y**, **anchor**, **text**) |
| php | function **drawText**( **$x**, **$y**, **$anchor**, **$text**) |
| cpp | int **drawText**( int **x**, int **y**, Y_ALIGN **anchor**, string **text**) |
| m | -(int) **drawText** : (int) **x**<br>: (int) **y**<br>: (Y_ALIGN) **anchor**<br>: (NSString*) **text** |
| pas | function **drawText**( **x**: LongInt,<br>**y**: LongInt,<br>**anchor**: TYALIGN,<br>**text**: string): LongInt |
| vb | function **drawText**( ) As Integer |
| cs | int **drawText**( int **x**, int **y**, ALIGN **anchor**, string **text**) |
| java | int **drawText**( int **x**, int **y**, ALIGN **anchor**, String **text**) |
| py | def **drawText**( **x**, **y**, **anchor**, **text**) |
| cmd | YDisplay **target** [-layer **layerId**] **drawText** **x y anchor text** |

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

**Parameters :**

**x** the distance from left of layer to the text anchor point, in pixels

**y** the distance from top of layer to the text anchor point, in pixels

**anchor** the text anchor point, chosen among the `Y_ALIGN` enumeration: `Y_ALIGN_TOP_LEFT`, `Y_ALIGN_CENTER_LEFT`, `Y_ALIGN_BASELINE_LEFT`, `Y_ALIGN_BOTTOM_LEFT`, `Y_ALIGN_TOP_CENTER`, `Y_ALIGN_CENTER`, `Y_ALIGN_BASELINE_CENTER`, `Y_ALIGN_BOTTOM_CENTER`, `Y_ALIGN_TOP_DECIMAL`, `Y_ALIGN_CENTER_DECIMAL`, `Y_ALIGN_BASELINE_DECIMAL`, `Y_ALIGN_BOTTOM_DECIMAL`, `Y_ALIGN_TOP_RIGHT`, `Y_ALIGN_CENTER_RIGHT`, `Y_ALIGN_BASELINE_RIGHT`, `Y_ALIGN_BOTTOM_RIGHT`.

**text** the text string to draw

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→get_display() displaylayer→display()[displaylayer display] YDisplayLayer

Gets parent YDisplay.

| | |
|---|---|
| js | function **get_display**( ) |
| nodejs | function **get_display**( ) |
| php | function **get_display**( ) |
| cpp | YDisplay* **get_display**( ) |
| m | -(YDisplay*) display |
| pas | function **get_display**( ): TYDisplay |
| vb | function **get_display**( ) As YDisplay |
| cs | YDisplay **get_display**( ) |
| java | YDisplay **get_display**( ) |
| py | def **get_display**( ) |

Returns the parent YDisplay object of the current YDisplayLayer.

**Returns :**
an `YDisplay` object

## displaylayer→get_displayHeight() displaylayer→displayHeight()[displaylayer displayHeight] YDisplayLayer

Returns the display height, in pixels.

| | |
|---|---|
| js | function **get_displayHeight**( ) |
| nodejs | function **get_displayHeight**( ) |
| php | function **get_displayHeight**( ) |
| cpp | int **get_displayHeight**( ) |
| m | -(int) displayHeight |
| pas | function **get_displayHeight**( ): LongInt |
| vb | function **get_displayHeight**( ) As Integer |
| cs | int **get_displayHeight**( ) |
| java | int **get_displayHeight**( ) |
| py | def **get_displayHeight**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **get_displayHeight** |

**Returns :**

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

## displaylayer→get_displayWidth() displaylayer→displayWidth()[displaylayer displayWidth] YDisplayLayer

Returns the display width, in pixels.

| | |
|---|---|
| js | function **get_displayWidth**( ) |
| nodejs | function **get_displayWidth**( ) |
| php | function **get_displayWidth**( ) |
| cpp | int **get_displayWidth**( ) |
| m | -(int) displayWidth |
| pas | function **get_displayWidth**( ): LongInt |
| vb | function **get_displayWidth**( ) As Integer |
| cs | int **get_displayWidth**( ) |
| java | int **get_displayWidth**( ) |
| py | def **get_displayWidth**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **get_displayWidth** |

**Returns :**

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

## displaylayer→get_layerHeight() displaylayer→layerHeight()[displaylayer layerHeight] YDisplayLayer

Returns the height of the layers to draw on, in pixels.

| | |
|---|---|
| js | function **get_layerHeight**( ) |
| nodejs | function **get_layerHeight**( ) |
| php | function **get_layerHeight**( ) |
| cpp | int **get_layerHeight**( ) |
| m | -(int) layerHeight |
| pas | function **get_layerHeight**( ): LongInt |
| vb | function **get_layerHeight**( ) As Integer |
| cs | int **get_layerHeight**( ) |
| java | int **get_layerHeight**( ) |
| py | def **get_layerHeight**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **get_layerHeight** |

**Returns :**
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

## displaylayer→get_layerWidth() displaylayer→layerWidth()[displaylayer layerWidth] YDisplayLayer

Returns the width of the layers to draw on, in pixels.

| | |
|---|---|
| js | function **get_layerWidth**( ) |
| nodejs | function **get_layerWidth**( ) |
| php | function **get_layerWidth**( ) |
| cpp | int **get_layerWidth**( ) |
| m | -(int) layerWidth |
| pas | function **get_layerWidth**( ): LongInt |
| vb | function **get_layerWidth**( ) As Integer |
| cs | int **get_layerWidth**( ) |
| java | int **get_layerWidth**( ) |
| py | def **get_layerWidth**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **hide**( ) |
| nodejs | function **hide**( ) |
| php | function **hide**( ) |
| cpp | int **hide**( ) |
| m | -(int) **hide** |
| pas | function **hide**( ): LongInt |
| vb | function **hide**( ) As Integer |
| cs | int **hide**( ) |
| java | int **hide**( ) |
| py | def **hide**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **lineTo**( **x**, **y**) |
| nodejs | function **lineTo**( **x**, **y**) |
| php | function **lineTo**( **$x**, **$y**) |
| cpp | int **lineTo**( int **x**, int **y**) |
| m | -(int) **lineTo** : (int) **x**<br>: (int) **y** |
| pas | function **lineTo**( **x**: LongInt, **y**: LongInt): LongInt |
| vb | function **lineTo**( ) As Integer |
| cs | int **lineTo**( int **x**, int **y**) |
| java | int **lineTo**( int **x**, int **y**) |
| py | def **lineTo**( **x**, **y**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **moveTo**( **x**, **y**) |
| nodejs | function **moveTo**( **x**, **y**) |
| php | function **moveTo**( **$x**, **$y**) |
| cpp | int **moveTo**( int **x**, int **y**) |
| m | -(int) **moveTo** : (int) **x** : (int) **y** |
| pas | function **moveTo**( **x**: LongInt, **y**: LongInt): LongInt |
| vb | function **moveTo**( ) As Integer |
| cs | int **moveTo**( int **x**, int **y**) |
| java | int **moveTo**( int **x**, int **y**) |
| py | def **moveTo**( **x**, **y**) |
| cmd | YDisplay **target** [-layer **layerId**] **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).

| | |
|---|---|
| js | function **reset**( ) |
| nodejs | function **reset**( ) |
| php | function **reset**( ) |
| cpp | int **reset**( ) |
| m | -(int) **reset** |
| pas | function **reset**( ): LongInt |
| vb | function **reset**( ) As Integer |
| cs | int **reset**( ) |
| java | int **reset**( ) |
| py | def **reset**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **reset** |

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

**Returns :**
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→selectColorPen()[displaylayer selectColorPen: ] YDisplayLayer

Selects the pen color for all subsequent drawing functions, including text drawing.

| | |
|---|---|
| js | function **selectColorPen**( **color**) |
| nodejs | function **selectColorPen**( **color**) |
| php | function **selectColorPen**( **$color**) |
| cpp | int **selectColorPen**( int **color**) |
| m | -(int) **selectColorPen** : (int) **color** |
| pas | function **selectColorPen**( **color**: LongInt): LongInt |
| vb | function **selectColorPen**( ) As Integer |
| cs | int **selectColorPen**( int **color**) |
| java | int **selectColorPen**( int **color**) |
| py | def **selectColorPen**( **color**) |
| cmd | YDisplay **target** [-layer **layerId**] **selectColorPen** **color** |

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

**Parameters :**

**color** the desired pen color, as a 24-bit RGB value

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→selectEraser()[displaylayer selectEraser] YDisplayLayer

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

| | |
|---|---|
| js | function **selectEraser**( ) |
| nodejs | function **selectEraser**( ) |
| php | function **selectEraser**( ) |
| cpp | int **selectEraser**( ) |
| m | -(int) **selectEraser** |
| pas | function **selectEraser**( ): LongInt |
| vb | function **selectEraser**( ) As Integer |
| cs | int **selectEraser**( ) |
| java | int **selectEraser**( ) |
| py | def **selectEraser**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **selectFont**( **fontname**) |
| nodejs | function **selectFont**( **fontname**) |
| php | function **selectFont**( **$fontname**) |
| cpp | int **selectFont**( string **fontname**) |
| m | -(int) **selectFont** : (NSString*) **fontname** |
| pas | function **selectFont**( **fontname**: string): LongInt |
| vb | function **selectFont**( ) As Integer |
| cs | int **selectFont**( string **fontname**) |
| java | int **selectFont**( String **fontname**) |
| py | def **selectFont**( **fontname**) |
| cmd | YDisplay **target** [-layer **layerId**] **selectFont** **fontname** |

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

**Parameters :**

**fontname** the font file name

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→selectGrayPen()[displaylayer selectGrayPen: ] YDisplayLayer

Selects the pen gray level for all subsequent drawing functions, including text drawing.

| | |
|---|---|
| js | function **selectGrayPen**( **graylevel**) |
| nodejs | function **selectGrayPen**( **graylevel**) |
| php | function **selectGrayPen**( **$graylevel**) |
| cpp | int **selectGrayPen**( int **graylevel**) |
| m | -(int) **selectGrayPen** : (int) **graylevel** |
| pas | function **selectGrayPen**( **graylevel**: LongInt): LongInt |
| vb | function **selectGrayPen**( ) As Integer |
| cs | int **selectGrayPen**( int **graylevel**) |
| java | int **selectGrayPen**( int **graylevel**) |
| py | def **selectGrayPen**( **graylevel**) |
| cmd | YDisplay **target** [-layer **layerId**] **selectGrayPen** **graylevel** |

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the lighest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

**Parameters :**

**graylevel** the desired gray level, from 0 to 255

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→setAntialiasingMode()[displaylayer setAntialiasingMode: ] YDisplayLayer

Enables or disables anti-aliasing for drawing oblique lines and circles.

| | |
|---|---|
| js | function **setAntialiasingMode**( **mode**) |
| nodejs | function **setAntialiasingMode**( **mode**) |
| php | function **setAntialiasingMode**( **$mode**) |
| cpp | int **setAntialiasingMode**( bool **mode**) |
| m | -(int) **setAntialiasingMode** : (bool) **mode** |
| pas | function **setAntialiasingMode**( **mode**: boolean): LongInt |
| vb | function **setAntialiasingMode**( ) As Integer |
| cs | int **setAntialiasingMode**( bool **mode**) |
| java | int **setAntialiasingMode**( boolean **mode**) |
| py | def **setAntialiasingMode**( **mode**) |
| cmd | YDisplay **target** [-layer **layerId**] **setAntialiasingMode** **mode** |

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

**Parameters :**

**mode** true to enable antialiasing, false to disable it.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→setConsoleBackground()[displaylayer setConsoleBackground: ] YDisplayLayer

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

| | |
|---|---|
| js | function **setConsoleBackground**( **bgcol**) |
| nodejs | function **setConsoleBackground**( **bgcol**) |
| php | function **setConsoleBackground**( **$bgcol**) |
| cpp | int **setConsoleBackground**( int **bgcol**) |
| m | -(int) **setConsoleBackground** : (int) **bgcol** |
| pas | function **setConsoleBackground**( **bgcol**: LongInt): LongInt |
| vb | function **setConsoleBackground**( ) As Integer |
| cs | int **setConsoleBackground**( int **bgcol**) |
| java | int **setConsoleBackground**( int **bgcol**) |
| py | def **setConsoleBackground**( **bgcol**) |
| cmd | YDisplay **target** [-layer **layerId**] **setConsoleBackground** **bgcol** |

**Parameters :**

**bgcol** the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→setConsoleMargins()[displaylayer setConsoleMargins: ] YDisplayLayer

Sets up display margins for the `consoleOut` function.

| | |
|---|---|
| js | function **setConsoleMargins**( **x1**, **y1**, **x2**, **y2**) |
| nodejs | function **setConsoleMargins**( **x1**, **y1**, **x2**, **y2**) |
| php | function **setConsoleMargins**( **$x1**, **$y1**, **$x2**, **$y2**) |
| cpp | int **setConsoleMargins**( int **x1**, int **y1**, int **x2**, int **y2**) |
| m | -(int) **setConsoleMargins** : (int) **x1** : (int) **y1** : (int) **x2** : (int) **y2** |
| pas | function **setConsoleMargins**( **x1**: LongInt, **y1**: LongInt, **x2**: LongInt, **y2**: LongInt): LongInt |
| vb | function **setConsoleMargins**( ) As Integer |
| cs | int **setConsoleMargins**( int **x1**, int **y1**, int **x2**, int **y2**) |
| java | int **setConsoleMargins**( int **x1**, int **y1**, int **x2**, int **y2**) |
| py | def **setConsoleMargins**( **x1**, **y1**, **x2**, **y2**) |
| cmd | YDisplay **target** [-layer **layerId**] **setConsoleMargins** **x1** **y1** **x2** **y2** |

**Parameters :**

- **x1** the distance from left of layer to the left margin, in pixels
- **y1** the distance from top of layer to the top margin, in pixels
- **x2** the distance from left of layer to the right margin, in pixels
- **y2** the distance from top of layer to the bottom margin, in pixels

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→setConsoleWordWrap()[displaylayer setConsoleWordWrap: ] YDisplayLayer

Sets up the wrapping behaviour used by the `consoleOut` function.

| | |
|---|---|
| js | function **setConsoleWordWrap**( **wordwrap**) |
| nodejs | function **setConsoleWordWrap**( **wordwrap**) |
| php | function **setConsoleWordWrap**( **$wordwrap**) |
| cpp | int **setConsoleWordWrap**( bool **wordwrap**) |
| m | -(int) **setConsoleWordWrap** : (bool) **wordwrap** |
| pas | function **setConsoleWordWrap**( **wordwrap**: boolean): LongInt |
| vb | function **setConsoleWordWrap**( ) As Integer |
| cs | int **setConsoleWordWrap**( bool **wordwrap**) |
| java | int **setConsoleWordWrap**( boolean **wordwrap**) |
| py | def **setConsoleWordWrap**( **wordwrap**) |
| cmd | YDisplay **target** [-layer **layerId**] **setConsoleWordWrap** **wordwrap** |

**Parameters :**

**wordwrap** `true` to wrap only between words, `false` to wrap on the last column anyway.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## displaylayer→setLayerPosition()[displaylayer setLayerPosition: ] YDisplayLayer

Sets the position of the layer relative to the display upper left corner.

| | |
|---|---|
| js | function **setLayerPosition**( **x**, **y**, **scrollTime**) |
| nodejs | function **setLayerPosition**( **x**, **y**, **scrollTime**) |
| php | function **setLayerPosition**( **$x**, **$y**, **$scrollTime**) |
| cpp | int **setLayerPosition**( int **x**, int **y**, int **scrollTime**) |
| m | -(int) **setLayerPosition** : (int) **x**<br>: (int) **y**<br>: (int) **scrollTime** |
| pas | function **setLayerPosition**( **x**: LongInt,<br>**y**: LongInt,<br>**scrollTime**: LongInt): LongInt |
| vb | function **setLayerPosition**( ) As Integer |
| cs | int **setLayerPosition**( int **x**, int **y**, int **scrollTime**) |
| java | int **setLayerPosition**( int **x**, int **y**, int **scrollTime**) |
| py | def **setLayerPosition**( **x**, **y**, **scrollTime**) |
| cmd | YDisplay **target** [-layer **layerId**] **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.

| | |
|---|---|
| js | function **unhide**( ) |
| nodejs | function **unhide**( ) |
| php | function **unhide**( ) |
| cpp | int **unhide**( ) |
| m | -(int) **unhide** |
| pas | function **unhide**( ): LongInt |
| vb | function **unhide**( ) As Integer |
| cs | int **unhide**( ) |
| java | int **unhide**( ) |
| py | def **unhide**( ) |
| cmd | YDisplay **target** [-layer **layerId**] **unhide** |

Shows the layer again after a hide command.

**Returns :**
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

# 3.15. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_dualpower.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YDualPower = yoctolib.YDualPower; |
| php | require_once('yocto_dualpower.php'); |
| cpp | #include "yocto_dualpower.h" |
| m | #import "yocto_dualpower.h" |
| pas | uses yocto_dualpower; |
| vb | yocto_dualpower.vb |
| cs | yocto_dualpower.cs |
| java | import com.yoctopuce.YoctoAPI.YDualPower; |
| py | from yocto_dualpower import * |

**Global functions**

**yFindDualPower**(**func**)

Retrieves a dual power control for a given identifier.

**yFirstDualPower**()

Starts the enumeration of dual power controls currently accessible.

**`YDualPower` methods**

**dualpower→describe**()

Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**dualpower→get_advertisedValue**()

Returns the current value of the power control (no more than 6 characters).

**dualpower→get_errorMessage**()

Returns the error message of the latest error with the power control.

**dualpower→get_errorType**()

Returns the numerical error code of the latest error with the power control.

**dualpower→get_extVoltage**()

Returns the measured voltage on the external power source, in millivolts.

**dualpower→get_friendlyName**()

Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.

**dualpower→get_functionDescriptor**()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**dualpower→get_functionId**()

Returns the hardware identifier of the power control, without reference to the module.

**dualpower→get_hardwareId**()

Returns the unique hardware identifier of the power control in the form `SERIAL.FUNCTIONID`.

**dualpower→get_logicalName**()

Returns the logical name of the power control.

**dualpower→get_module**()

Gets the `YModule` object for the device on which the function is located.

**dualpower→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**dualpower→get_powerControl()**

Returns the selected power source for module functions that require lots of current.

**dualpower→get_powerState()**

Returns the current power source for module functions that require lots of current.

**dualpower→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**dualpower→isOnline()**

Checks if the power control is currently reachable, without raising any error.

**dualpower→isOnline_async(callback, context)**

Checks if the power control is currently reachable, without raising any error (asynchronous version).

**dualpower→load(msValidity)**

Preloads the power control cache with a specified validity duration.

**dualpower→load_async(msValidity, callback, context)**

Preloads the power control cache with a specified validity duration (asynchronous version).

**dualpower→nextDualPower()**

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

**dualpower→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**dualpower→set_logicalName(newval)**

Changes the logical name of the power control.

**dualpower→set_powerControl(newval)**

Changes the selected power source for module functions that require lots of current.

**dualpower→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**dualpower→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDualPower.FindDualPower() yFindDualPower()yFindDualPower() YDualPower

Retrieves a dual power control for a given identifier.

| | |
|---|---|
| js | function **yFindDualPower**( **func**) |
| nodejs | function **FindDualPower**( **func**) |
| php | function **yFindDualPower**( **$func**) |
| cpp | YDualPower* **yFindDualPower**( const string& **func**) |
| m | YDualPower* **yFindDualPower**( NSString* **func**) |
| pas | function **yFindDualPower**( **func**: string): TYDualPower |
| vb | function **yFindDualPower**( ByVal **func** As String) As YDualPower |
| cs | YDualPower **FindDualPower**( string **func**) |
| java | YDualPower **FindDualPower**( String **func**) |
| py | def **FindDualPower**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDualPower.isOnline()` to test if the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the power control

**Returns :**

a `YDualPower` object allowing you to drive the power control.

## YDualPower.FirstDualPower() yFirstDualPower()yFirstDualPower() YDualPower

Starts the enumeration of dual power controls currently accessible.

| | |
|---|---|
| js | function **yFirstDualPower**( ) |
| nodejs | function **FirstDualPower**( ) |
| php | function **yFirstDualPower**( ) |
| cpp | YDualPower* **yFirstDualPower**( ) |
| m | YDualPower* **yFirstDualPower**( ) |
| pas | function **yFirstDualPower**( ): TYDualPower |
| vb | function **yFirstDualPower**( ) As YDualPower |
| cs | YDualPower **FirstDualPower**( ) |
| java | YDualPower **FirstDualPower**( ) |
| py | def **FirstDualPower**( ) |

Use the method `YDualPower.nextDualPower()` to iterate on next dual power controls.

**Returns :**

a pointer to a `YDualPower` object, corresponding to the first dual power control currently online, or a `null` pointer if there are none.

## dualpower→describe()[dualpower describe] YDualPower

Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the power control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## dualpower→get_advertisedValue() dualpower→advertisedValue()[dualpower advertisedValue] YDualPower

Returns the current value of the power control (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YDualPower **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the power control (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## dualpower→get_errorMessage() dualpower→errorMessage()[dualpower errorMessage] YDualPower

Returns the error message of the latest error with the power control.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the power control object

## dualpower→get_errorType() dualpower→errorType() YDualPower

Returns the numerical error code of the latest error with the power control.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the power control object

## dualpower→get_extVoltage() dualpower→extVoltage()[dualpower extVoltage] YDualPower

Returns the measured voltage on the external power source, in millivolts.

| | |
|---|---|
| js | function **get_extVoltage**( ) |
| nodejs | function **get_extVoltage**( ) |
| php | function **get_extVoltage**( ) |
| cpp | int **get_extVoltage**( ) |
| m | -(int) extVoltage |
| pas | function **get_extVoltage**( ): LongInt |
| vb | function **get_extVoltage**( ) As Integer |
| cs | int **get_extVoltage**( ) |
| java | int **get_extVoltage**( ) |
| py | def **get_extVoltage**( ) |
| cmd | YDualPower **target** **get_extVoltage** |

**Returns :**
an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns `Y_EXTVOLTAGE_INVALID`.

## dualpower→get_friendlyName() dualpower→friendlyName()[dualpower friendlyName]

**YDualPower**

Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the power control if they are defined, otherwise the serial number of the module and the hardware identifier of the power control (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the power control using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## dualpower→get_functionDescriptor() dualpower→functionDescriptor()[dualpower functionDescriptor] YDualPower

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**
an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## dualpower→get_functionId() dualpower→functionId()[dualpower functionId] YDualPower

Returns the hardware identifier of the power control, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the power control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## dualpower→get_hardwareId() dualpower→hardwareId()[dualpower hardwareId] YDualPower

Returns the unique hardware identifier of the power control in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the power control. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the power control (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## dualpower→get_logicalName() dualpower→logicalName()[dualpower logicalName] YDualPower

Returns the logical name of the power control.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YDualPower **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the power control. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## dualpower→get_module() dualpower→module()[dualpower module] YDualPower

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## dualpower→get_module_async() dualpower→module_async() YDualPower

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## dualpower→get_powerControl() dualpower→powerControl()[dualpower powerControl] YDualPower

Returns the selected power source for module functions that require lots of current.

| | |
|---|---|
| js | function **get_powerControl**( ) |
| nodejs | function **get_powerControl**( ) |
| php | function **get_powerControl**( ) |
| cpp | Y_POWERCONTROL_enum **get_powerControl**( ) |
| m | -(Y_POWERCONTROL_enum) powerControl |
| pas | function **get_powerControl**( ): Integer |
| vb | function **get_powerControl**( ) As Integer |
| cs | int **get_powerControl**( ) |
| java | int **get_powerControl**( ) |
| py | def **get_powerControl**( ) |
| cmd | YDualPower **target** **get_powerControl** |

**Returns :**

a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns `Y_POWERCONTROL_INVALID`.

## dualpower→get_powerState() dualpower→powerState()[dualpower powerState] YDualPower

Returns the current power source for module functions that require lots of current.

| | |
|---|---|
| js | function **get_powerState**( ) |
| nodejs | function **get_powerState**( ) |
| php | function **get_powerState**( ) |
| cpp | Y_POWERSTATE_enum **get_powerState**( ) |
| m | -(Y_POWERSTATE_enum) powerState |
| pas | function **get_powerState**( ): Integer |
| vb | function **get_powerState**( ) As Integer |
| cs | int **get_powerState**( ) |
| java | int **get_powerState**( ) |
| py | def **get_powerState**( ) |
| cmd | YDualPower **target** **get_powerState** |

**Returns :**

a value among `Y_POWERSTATE_OFF`, `Y_POWERSTATE_FROM_USB` and `Y_POWERSTATE_FROM_EXT` corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns `Y_POWERSTATE_INVALID`.

## dualpower→get_userData() dualpower→userData()[dualpower userData] YDualPower

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## dualpower→isOnline()[dualpower isOnline] YDualPower

Checks if the power control is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power control.

**Returns :**

`true` if the power control can be reached, and `false` otherwise

## dualpower→isOnline_async() YDualPower

Checks if the power control is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## dualpower→load()[dualpower load: ] YDualPower

Preloads the power control cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## dualpower→load_async() YDualPower

Preloads the power control cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## dualpower→nextDualPower()[dualpower nextDualPower] YDualPower

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

| | |
|---|---|
| js | function **nextDualPower**( ) |
| nodejs | function **nextDualPower**( ) |
| php | function **nextDualPower**( ) |
| cpp | YDualPower * **nextDualPower**( ) |
| m | -(YDualPower*) **nextDualPower** |
| pas | function **nextDualPower**( ): TYDualPower |
| vb | function **nextDualPower**( ) As YDualPower |
| cs | YDualPower **nextDualPower**( ) |
| java | YDualPower **nextDualPower**( ) |
| py | def **nextDualPower**( ) |

**Returns :**

a pointer to a `YDualPower` object, corresponding to a dual power control currently online, or a `null` pointer if there are no more dual power controls to enumerate.

## dualpower→registerValueCallback()[dualpower registerValueCallback: ] YDualPower

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YDualPowerValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YDualPowerValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYDualPowerValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## dualpower→set_logicalName() dualpower→setLogicalName()[dualpower setLogicalName: ] YDualPower

Changes the logical name of the power control.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YDualPower **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the power control.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## dualpower→set_powerControl() dualpower→setPowerControl()[dualpower setPowerControl: ] YDualPower

Changes the selected power source for module functions that require lots of current.

| | |
|---|---|
| js | function **set_powerControl**( **newval**) |
| nodejs | function **set_powerControl**( **newval**) |
| php | function **set_powerControl**( **$newval**) |
| cpp | int **set_powerControl**( Y_POWERCONTROL_enum **newval**) |
| m | -(int) setPowerControl : (Y_POWERCONTROL_enum) **newval** |
| pas | function **set_powerControl**( **newval**: Integer): integer |
| vb | function **set_powerControl**( ByVal **newval** As Integer) As Integer |
| cs | int **set_powerControl**( int **newval**) |
| java | int **set_powerControl**( int **newval**) |
| py | def **set_powerControl**( **newval**) |
| cmd | YDualPower **target** **set_powerControl** **newval** |

**Parameters :**

**newval** a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## dualpower→set_userData() dualpower→setUserData()[dualpower setUserData: ] YDualPower

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## dualpower→wait_async() YDualPower

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.16. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_files.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YFiles = yoctolib.YFiles; |
| php | require_once('yocto_files.php'); |
| cpp | #include "yocto_files.h" |
| m | #import "yocto_files.h" |
| pas | uses yocto_files; |
| vb | yocto_files.vb |
| cs | yocto_files.cs |
| java | import com.yoctopuce.YoctoAPI.YFiles; |
| py | from yocto_files import * |

**Global functions**

**yFindFiles(func)**

Retrieves a filesystem for a given identifier.

**yFirstFiles()**

Starts the enumeration of filesystems currently accessible.

**YFiles methods**

**files→describe()**

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**files→download(pathname)**

Downloads the requested file and returns a binary buffer with its content.

**files→download_async(pathname, callback, context)**

Downloads the requested file and returns a binary buffer with its content.

**files→format_fs()**

Reinitializes the filesystem to its clean, unfragmented, empty state.

**files→get_advertisedValue()**

Returns the current value of the filesystem (no more than 6 characters).

**files→get_errorMessage()**

Returns the error message of the latest error with the filesystem.

**files→get_errorType()**

Returns the numerical error code of the latest error with the filesystem.

**files→get_filesCount()**

Returns the number of files currently loaded in the filesystem.

**files→get_freeSpace()**

Returns the free space for uploading new files to the filesystem, in bytes.

**files→get_friendlyName()**

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

**files→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**files→get_functionId()**

Returns the hardware identifier of the filesystem, without reference to the module.

**files→get_hardwareId()**

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

**files→get_list(pattern)**

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

**files→get_logicalName()**

Returns the logical name of the filesystem.

**files→get_module()**

Gets the `YModule` object for the device on which the function is located.

**files→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**files→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**files→isOnline()**

Checks if the filesystem is currently reachable, without raising any error.

**files→isOnline_async(callback, context)**

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

**files→load(msValidity)**

Preloads the filesystem cache with a specified validity duration.

**files→load_async(msValidity, callback, context)**

Preloads the filesystem cache with a specified validity duration (asynchronous version).

**files→nextFiles()**

Continues the enumeration of filesystems started using `yFirstFiles()`.

**files→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**files→remove(pathname)**

Deletes a file, given by its full path name, from the filesystem.

**files→set_logicalName(newval)**

Changes the logical name of the filesystem.

**files→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**files→upload(pathname, content)**

Uploads a file to the filesystem, to the specified full path name.

**files→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YFiles.FindFiles() yFindFiles()yFindFiles() YFiles

Retrieves a filesystem for a given identifier.

| | |
|---|---|
| js | function **yFindFiles**( **func**) |
| nodejs | function **FindFiles**( **func**) |
| php | function **yFindFiles**( **$func**) |
| cpp | YFiles* **yFindFiles**( string **func**) |
| m | +(YFiles*) **yFindFiles** : (NSString*) **func** |
| pas | function **yFindFiles**( **func**: string): TYFiles |
| vb | function **yFindFiles**( ByVal **func** As String) As YFiles |
| cs | YFiles **FindFiles**( string **func**) |
| java | YFiles **FindFiles**( String **func**) |
| py | def **FindFiles**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the filesystem

**Returns :**

a `YFiles` object allowing you to drive the filesystem.

## YFiles.FirstFiles() yFirstFiles()yFirstFiles()

YFiles

Starts the enumeration of filesystems currently accessible.

| | |
|---|---|
| js | function **yFirstFiles**( ) |
| nodejs | function **FirstFiles**( ) |
| php | function **yFirstFiles**( ) |
| cpp | YFiles* **yFirstFiles**( ) |
| m | YFiles* **yFirstFiles**( ) |
| pas | function **yFirstFiles**( ): TYFiles |
| vb | function **yFirstFiles**( ) As YFiles |
| cs | YFiles **FirstFiles**( ) |
| java | YFiles **FirstFiles**( ) |
| py | def **FirstFiles**( ) |

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

**Returns :**

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

## files→describe()[files describe] YFiles

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the 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.

| | |
|---|---|
| js | function **download**( **pathname**) |
| nodejs | function **download**( **pathname**) |
| php | function **download**( **$pathname**) |
| cpp | string **download**( string **pathname**) |
| m | -(NSData*) **download** : (NSString*) **pathname** |
| pas | function **download**( **pathname**: string): TByteArray |
| vb | function **download**( ) As Byte |
| py | def **download**( **pathname**) |
| cmd | YFiles **target** **download** **pathname** |

**Parameters :**

**pathname** path and name of the file to download

**Returns :**

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

## files→download_async()    YFiles

Downloads the requested file and returns a binary buffer with its content.

| | |
|---|---|
| js | function **download_async**( **pathname**, **callback**, **context**) |
| nodejs | function **download_async**( **pathname**, **callback**, **context**) |

This is the asynchronous version that uses a callback to pass the result when the donwload is completed.

**Parameters :**

**pathname** path and name of the new file to load

**callback** callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YFiles object whose download_async was invoked - a binary buffer with the file content

**context** user-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## files→format_fs()[files format_fs] YFiles

Reinitializes the filesystem to its clean, unfragmented, empty state.

| | |
|---|---|
| js | function **format_fs**( ) |
| nodejs | function **format_fs**( ) |
| php | function **format_fs**( ) |
| cpp | int **format_fs**( ) |
| m | -(int) **format_fs** |
| pas | function **format_fs**( ): LongInt |
| vb | function **format_fs**( ) As Integer |
| cs | int **format_fs**( ) |
| java | int **format_fs**( ) |
| py | def **format_fs**( ) |
| cmd | YFiles **target** **format_fs** |

All files previously uploaded are permanently lost.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## files→get_advertisedValue() files→advertisedValue()[files advertisedValue] YFiles

Returns the current value of the filesystem (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YFiles **target** **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()<br>files→errorMessage()[files errorMessage] — YFiles

Returns the error message of the latest error with the filesystem.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the filesystem object

## files→get_errorType()
## files→errorType()

**YFiles**

Returns the numerical error code of the latest error with the filesystem.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the filesystem object

## files→get_filesCount() files→filesCount()[files filesCount] YFiles

Returns the number of files currently loaded in the filesystem.

| | |
|---|---|
| js | function **get_filesCount**( ) |
| nodejs | function **get_filesCount**( ) |
| php | function **get_filesCount**( ) |
| cpp | int **get_filesCount**( ) |
| m | -(int) filesCount |
| pas | function **get_filesCount**( ): LongInt |
| vb | function **get_filesCount**( ) As Integer |
| cs | int **get_filesCount**( ) |
| java | int **get_filesCount**( ) |
| py | def **get_filesCount**( ) |
| cmd | YFiles **target** **get_filesCount** |

**Returns :**
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `Y_FILESCOUNT_INVALID`.

## files→get_freeSpace() files→freeSpace()[files freeSpace] YFiles

Returns the free space for uploading new files to the filesystem, in bytes.

| | |
|---|---|
| js | function **get_freeSpace**( ) |
| nodejs | function **get_freeSpace**( ) |
| php | function **get_freeSpace**( ) |
| cpp | int **get_freeSpace**( ) |
| m | -(int) freeSpace |
| pas | function **get_freeSpace**( ): LongInt |
| vb | function **get_freeSpace**( ) As Integer |
| cs | int **get_freeSpace**( ) |
| java | int **get_freeSpace**( ) |
| py | def **get_freeSpace**( ) |
| cmd | YFiles **target** **get_freeSpace** |

**Returns :**
an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns `Y_FREESPACE_INVALID`.

## files→get_friendlyName() files→friendlyName()[files friendlyName] YFiles

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## files→get_functionDescriptor() files→functionDescriptor()[files functionDescriptor] YFiles

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## files→get_functionId()<br>files→functionId()[files functionId] **YFiles**

Returns the hardware identifier of the filesystem, without reference to the module.

| | |
|---|---|
| `js` | function **get_functionId**( ) |
| `nodejs` | function **get_functionId**( ) |
| `php` | function **get_functionId**( ) |
| `cpp` | string **get_functionId**( ) |
| `m` | -(NSString*) functionId |
| `vb` | function **get_functionId**( ) As String |
| `cs` | string **get_functionId**( ) |
| `java` | String **get_functionId**( ) |
| `py` | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the filesystem (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## files→get_hardwareId() files→hardwareId()[files hardwareId] YFiles

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the filesystem (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## files→get_list() files→list()[files list: ] YFiles

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

| | |
|---|---|
| js | function **get_list**( **pattern**) |
| nodejs | function **get_list**( **pattern**) |
| php | function **get_list**( **$pattern**) |
| cpp | vector<YFileRecord> **get_list**( string **pattern**) |
| m | -(NSMutableArray*) list : (NSString*) **pattern** |
| pas | function **get_list**( **pattern**: string): TYFileRecordArray |
| vb | function **get_list**( ) As List |
| cs | List<YFileRecord> **get_list**( string **pattern**) |
| java | ArrayList<YFileRecord> **get_list**( String **pattern**) |
| py | def **get_list**( **pattern**) |
| cmd | YFiles **target** **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 logicalName]

Returns the logical name of the filesystem.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YFiles **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the filesystem. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## files→get_module() files→module()[files module] YFiles

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## files→get_module_async() files→module_async() YFiles

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## files→get_userData() files→userData()[files userData] YFiles

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## files→isOnline()[files isOnline] YFiles

Checks if the filesystem is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

**Returns :**
`true` if the filesystem can be reached, and `false` otherwise

## files→isOnline_async() YFiles

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## files→load()[files load: ] YFiles

Preloads the filesystem cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## files→load_async() YFiles

Preloads the filesystem cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## files→nextFiles()[files nextFiles] YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

| | |
|---|---|
| js | function **nextFiles**( ) |
| nodejs | function **nextFiles**( ) |
| php | function **nextFiles**( ) |
| cpp | YFiles * **nextFiles**( ) |
| m | -(YFiles*) **nextFiles** |
| pas | function **nextFiles**( ): TYFiles |
| vb | function **nextFiles**( ) As YFiles |
| cs | YFiles **nextFiles**( ) |
| java | YFiles **nextFiles**( ) |
| py | def **nextFiles**( ) |

**Returns :**

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

## files→registerValueCallback()[files registerValueCallback: ] YFiles

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YFilesValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YFilesValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYFilesValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## files→remove()[files remove: ] YFiles

Deletes a file, given by its full path name, from the filesystem.

| | |
|---|---|
| js | function **remove**( **pathname**) |
| nodejs | function **remove**( **pathname**) |
| php | function **remove**( **$pathname**) |
| cpp | int **remove**( string **pathname**) |
| m | -(int) **remove** : (NSString*) **pathname** |
| pas | function **remove**( **pathname**: string): LongInt |
| vb | function **remove**( ) As Integer |
| cs | int **remove**( string **pathname**) |
| java | int **remove**( String **pathname**) |
| py | def **remove**( **pathname**) |
| cmd | YFiles **target** **remove** **pathname** |

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

**Parameters :**

**pathname** path and name of the file to remove.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## files→set_logicalName() files→setLogicalName()[files setLogicalName: ] YFiles

Changes the logical name of the filesystem.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YFiles **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the 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 setUserData: ]

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## files→upload()[files upload: ] YFiles

Uploads a file to the filesystem, to the specified full path name.

| | |
|---|---|
| `js` | function **upload**( **pathname**, **content**) |
| `nodejs` | function **upload**( **pathname**, **content**) |
| `php` | function **upload**( **$pathname**, **$content**) |
| `cpp` | int **upload**( string **pathname**, string **content**) |
| `m` | -(int) **upload** : (NSString*) **pathname**<br>: (NSData*) **content** |
| `pas` | function **upload**( **pathname**: string, **content**: TByteArray): LongInt |
| `vb` | procedure **upload**( ) |
| `cs` | int **upload**( string **pathname**) |
| `java` | int **upload**( String **pathname**) |
| `py` | def **upload**( **pathname**, **content**) |
| `cmd` | YFiles **target** **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() YFiles

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js function **wait_async**( **callback**, **context**)
nodejs function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.17. GenericSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_genericsensor.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YGenericSensor = yoctolib.YGenericSensor; |
| php | require_once('yocto_genericsensor.php'); |
| cpp | #include "yocto_genericsensor.h" |
| m | #import "yocto_genericsensor.h" |
| pas | uses yocto_genericsensor; |
| vb | yocto_genericsensor.vb |
| cs | yocto_genericsensor.cs |
| java | import com.yoctopuce.YoctoAPI.YGenericSensor; |
| py | from yocto_genericsensor import * |

**Global functions**

**yFindGenericSensor(func)**

Retrieves a generic sensor for a given identifier.

**yFirstGenericSensor()**

Starts the enumeration of generic sensors currently accessible.

**`YGenericSensor` methods**

**genericsensor→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**genericsensor→describe()**

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**genericsensor→get_advertisedValue()**

Returns the current value of the generic sensor (no more than 6 characters).

**genericsensor→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**genericsensor→get_currentValue()**

Returns the current measured value.

**genericsensor→get_errorMessage()**

Returns the error message of the latest error with the generic sensor.

**genericsensor→get_errorType()**

Returns the numerical error code of the latest error with the generic sensor.

**genericsensor→get_friendlyName()**

Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**genericsensor→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**genericsensor→get_functionId()**

Returns the hardware identifier of the generic sensor, without reference to the module.

**genericsensor→get_hardwareId()**

Returns the unique hardware identifier of the generic sensor in the form `SERIAL.FUNCTIONID`.

**genericsensor→get_highestValue()**

Returns the maximal value observed for the measure since the device was started.

**genericsensor→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**genericsensor→get_logicalName()**

Returns the logical name of the generic sensor.

**genericsensor→get_lowestValue()**

Returns the minimal value observed for the measure since the device was started.

**genericsensor→get_module()**

Gets the `YModule` object for the device on which the function is located.

**genericsensor→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**genericsensor→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**genericsensor→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**genericsensor→get_resolution()**

Returns the resolution of the measured values.

**genericsensor→get_signalRange()**

Returns the electric signal range used by the sensor.

**genericsensor→get_signalUnit()**

Returns the measuring unit of the electrical signal used by the sensor.

**genericsensor→get_signalValue()**

Returns the measured value of the electrical signal used by the sensor.

**genericsensor→get_unit()**

Returns the measuring unit for the measure.

**genericsensor→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**genericsensor→get_valueRange()**

Returns the physical value range measured by the sensor.

**genericsensor→isOnline()**

Checks if the generic sensor is currently reachable, without raising any error.

**genericsensor→isOnline_async(callback, context)**

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

**genericsensor→load(msValidity)**

Preloads the generic sensor cache with a specified validity duration.

**genericsensor→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**genericsensor→load_async(msValidity, callback, context)**

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

**genericsensor→nextGenericSensor()**

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

**genericsensor→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**genericsensor→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**genericsensor→set_highestValue(newval)**

Changes the recorded maximal value observed.

**genericsensor→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**genericsensor→set_logicalName(newval)**

Changes the logical name of the generic sensor.

**genericsensor→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**genericsensor→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**genericsensor→set_resolution(newval)**

Changes the resolution of the measured physical values.

**genericsensor→set_signalRange(newval)**

Changes the electric signal range used by the sensor.

**genericsensor→set_unit(newval)**

Changes the measuring unit for the measured value.

**genericsensor→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**genericsensor→set_valueRange(newval)**

Changes the physical value range measured by the sensor.

**genericsensor→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YGenericSensor.FindGenericSensor() yFindGenericSensor()yFindGenericSensor()

**YGenericSensor**

Retrieves a generic sensor for a given identifier.

| | |
|---|---|
| js | function **yFindGenericSensor**( **func**) |
| nodejs | function **FindGenericSensor**( **func**) |
| php | function **yFindGenericSensor**( **$func**) |
| cpp | YGenericSensor* **yFindGenericSensor**( const string& **func**) |
| m | YGenericSensor* **yFindGenericSensor**( NSString* **func**) |
| pas | function **yFindGenericSensor**( **func**: string): TYGenericSensor |
| vb | function **yFindGenericSensor**( ByVal **func** As String) As YGenericSensor |
| cs | YGenericSensor **FindGenericSensor**( string **func**) |
| java | YGenericSensor **FindGenericSensor**( String **func**) |
| py | def **FindGenericSensor**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the generic sensor

**Returns :**

a `YGenericSensor` object allowing you to drive the generic sensor.

## YGenericSensor.FirstGenericSensor() yFirstGenericSensor()yFirstGenericSensor() YGenericSensor

Starts the enumeration of generic sensors currently accessible.

| | |
|---|---|
| js | function **yFirstGenericSensor**( ) |
| nodejs | function **FirstGenericSensor**( ) |
| php | function **yFirstGenericSensor**( ) |
| cpp | YGenericSensor* **yFirstGenericSensor**( ) |
| m | YGenericSensor* **yFirstGenericSensor**( ) |
| pas | function **yFirstGenericSensor**( ): TYGenericSensor |
| vb | function **yFirstGenericSensor**( ) As YGenericSensor |
| cs | YGenericSensor **FirstGenericSensor**( ) |
| java | YGenericSensor **FirstGenericSensor**( ) |
| py | def **FirstGenericSensor**( ) |

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

**Returns :**

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a `null` pointer if there are none.

## genericsensor→calibrateFromPoints()[genericsensor calibrateFromPoints: ] YGenericSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YGenericSensor **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→describe()[genericsensor describe] YGenericSensor

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the generic sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## genericsensor→get_advertisedValue() genericsensor→advertisedValue()[genericsensor advertisedValue] YGenericSensor

Returns the current value of the generic sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YGenericSensor **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the generic sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## genericsensor→get_currentRawValue() genericsensor→currentRawValue()[genericsensor currentRawValue] YGenericSensor

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YGenericSensor **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## genericsensor→get_currentValue() genericsensor→currentValue()[genericsensor currentValue] YGenericSensor

Returns the current measured value.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YGenericSensor **target** **get_currentValue** |

**Returns :**
a floating point number corresponding to the current measured value

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## genericsensor→get_errorMessage() genericsensor→errorMessage()[genericsensor errorMessage] YGenericSensor

Returns the error message of the latest error with the generic sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the generic sensor object

## genericsensor→get_errorType() genericsensor→errorType() YGenericSensor

Returns the numerical error code of the latest error with the generic sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the generic sensor object

## genericsensor→get_friendlyName() genericsensor→friendlyName()[genericsensor friendlyName] YGenericSensor

Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the generic sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the generic sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the generic sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## genericsensor→get_functionDescriptor() genericsensor→functionDescriptor()[genericsensor functionDescriptor] YGenericSensor

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## genericsensor→get_functionId() genericsensor→functionId()[genericsensor functionId] YGenericSensor

Returns the hardware identifier of the generic sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the generic sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## genericsensor→get_hardwareId() genericsensor→hardwareId()[genericsensor hardwareId] YGenericSensor

Returns the unique hardware identifier of the generic sensor in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the generic sensor. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the generic sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## genericsensor→get_highestValue() genericsensor→highestValue()[genericsensor highestValue] YGenericSensor

Returns the maximal value observed for the measure since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YGenericSensor **target** **get_highestValue** |

**Returns :**
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## genericsensor→get_logFrequency() genericsensor→logFrequency()[genericsensor logFrequency] YGenericSensor

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YGenericSensor **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## genericsensor→get_logicalName() genericsensor→logicalName()[genericsensor logicalName] YGenericSensor

Returns the logical name of the generic sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YGenericSensor **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the generic sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## genericsensor→get_lowestValue() genericsensor→lowestValue()[genericsensor lowestValue] YGenericSensor

Returns the minimal value observed for the measure since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YGenericSensor **target** **get_lowestValue** |

**Returns :**

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## genericsensor→get_module() genericsensor→module()[genericsensor module] YGenericSensor

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## genericsensor→get_module_async() genericsensor→module_async() YGenericSensor

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## genericsensor→get_recordedData() genericsensor→recordedData()[genericsensor recordedData: ] YGenericSensor

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YGenericSensor **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## genericsensor→get_reportFrequency() genericsensor→reportFrequency()[genericsensor reportFrequency] YGenericSensor

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YGenericSensor **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## genericsensor→get_resolution() genericsensor→resolution()[genericsensor resolution] YGenericSensor

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YGenericSensor **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## genericsensor→get_signalRange() genericsensor→signalRange()[genericsensor signalRange] YGenericSensor

Returns the electric signal range used by the sensor.

| | |
|---|---|
| js | function **get_signalRange**( ) |
| nodejs | function **get_signalRange**( ) |
| php | function **get_signalRange**( ) |
| cpp | string **get_signalRange**( ) |
| m | -(NSString*) signalRange |
| pas | function **get_signalRange**( ): string |
| vb | function **get_signalRange**( ) As String |
| cs | string **get_signalRange**( ) |
| java | String **get_signalRange**( ) |
| py | def **get_signalRange**( ) |
| cmd | YGenericSensor **target** **get_signalRange** |

**Returns :**
a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns `Y_SIGNALRANGE_INVALID`.

## genericsensor→get_signalUnit() genericsensor→signalUnit()[genericsensor signalUnit]  YGenericSensor

Returns the measuring unit of the electrical signal used by the sensor.

| | |
|---|---|
| js | function **get_signalUnit**( ) |
| nodejs | function **get_signalUnit**( ) |
| php | function **get_signalUnit**( ) |
| cpp | string **get_signalUnit**( ) |
| m | -(NSString*) signalUnit |
| pas | function **get_signalUnit**( ): string |
| vb | function **get_signalUnit**( ) As String |
| cs | string **get_signalUnit**( ) |
| java | String **get_signalUnit**( ) |
| py | def **get_signalUnit**( ) |
| cmd | YGenericSensor **target** **get_signalUnit** |

**Returns :**
a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns `Y_SIGNALUNIT_INVALID`.

## genericsensor→get_signalValue() genericsensor→signalValue()[genericsensor signalValue] YGenericSensor

Returns the measured value of the electrical signal used by the sensor.

| | |
|---|---|
| js | function **get_signalValue**( ) |
| nodejs | function **get_signalValue**( ) |
| php | function **get_signalValue**( ) |
| cpp | double **get_signalValue**( ) |
| m | -(double) signalValue |
| pas | function **get_signalValue**( ): double |
| vb | function **get_signalValue**( ) As Double |
| cs | double **get_signalValue**( ) |
| java | double **get_signalValue**( ) |
| py | def **get_signalValue**( ) |
| cmd | YGenericSensor **target** **get_signalValue** |

**Returns :**

a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns `Y_SIGNALVALUE_INVALID`.

## genericsensor→get_unit() genericsensor→unit()[genericsensor unit] YGenericSensor

Returns the measuring unit for the measure.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YGenericSensor **target** **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 userData]

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## genericsensor→get_valueRange() genericsensor→valueRange()[genericsensor valueRange] YGenericSensor

Returns the physical value range measured by the sensor.

| | |
|---|---|
| js | function **get_valueRange**( ) |
| nodejs | function **get_valueRange**( ) |
| php | function **get_valueRange**( ) |
| cpp | string **get_valueRange**( ) |
| m | -(NSString*) valueRange |
| pas | function **get_valueRange**( ): string |
| vb | function **get_valueRange**( ) As String |
| cs | string **get_valueRange**( ) |
| java | String **get_valueRange**( ) |
| py | def **get_valueRange**( ) |
| cmd | YGenericSensor **target** **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.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the generic sensor.

**Returns :**

`true` if the generic sensor can be reached, and `false` otherwise

## genericsensor→isOnline_async() YGenericSensor

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## genericsensor→load()[genericsensor load: ] YGenericSensor

Preloads the generic sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## genericsensor→loadCalibrationPoints() [genericsensor loadCalibrationPoints: ] YGenericSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YGenericSensor **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→load_async() YGenericSensor

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## genericsensor→nextGenericSensor()[genericsensor nextGenericSensor] YGenericSensor

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

| | |
|---|---|
| js | function **nextGenericSensor**( ) |
| nodejs | function **nextGenericSensor**( ) |
| php | function **nextGenericSensor**( ) |
| cpp | YGenericSensor * **nextGenericSensor**( ) |
| m | -(YGenericSensor*) **nextGenericSensor** |
| pas | function **nextGenericSensor**( ): TYGenericSensor |
| vb | function **nextGenericSensor**( ) As YGenericSensor |
| cs | YGenericSensor **nextGenericSensor**( ) |
| java | YGenericSensor **nextGenericSensor**( ) |
| py | def **nextGenericSensor**( ) |

**Returns :**

a pointer to a `YGenericSensor` object, corresponding to a generic sensor currently online, or a `null` pointer if there are no more generic sensors to enumerate.

## genericsensor→registerTimedReportCallback() [genericsensor registerTimedReportCallback: ] YGenericSensor

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YGenericSensorTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YGenericSensorTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYGenericSensorTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## genericsensor→registerValueCallback() [genericsensor registerValueCallback: ]

**YGenericSensor**

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YGenericSensorValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YGenericSensorValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYGenericSensorValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## genericsensor→set_highestValue() genericsensor→setHighestValue()[genericsensor setHighestValue: ] YGenericSensor

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YGenericSensor **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→set_logFrequency() genericsensor→setLogFrequency()[genericsensor setLogFrequency: ] YGenericSensor

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YGenericSensor **target set_logFrequency newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→set_logicalName() genericsensor→setLogicalName()[genericsensor setLogicalName: ] YGenericSensor

Changes the logical name of the generic sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YGenericSensor **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the generic sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## genericsensor→set_lowestValue() genericsensor→setLowestValue()[genericsensor setLowestValue: ] YGenericSensor

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YGenericSensor **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→set_reportFrequency() genericsensor→setReportFrequency() [genericsensor setReportFrequency: ]

**YGenericSensor**

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YGenericSensor **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→set_resolution() genericsensor→setResolution()[genericsensor setResolution: ] — YGenericSensor

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YGenericSensor **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→set_signalRange() genericsensor→setSignalRange()[genericsensor setSignalRange: ] YGenericSensor

Changes the electric signal range used by the sensor.

| | |
|---|---|
| js | function **set_signalRange**( **newval**) |
| nodejs | function **set_signalRange**( **newval**) |
| php | function **set_signalRange**( **$newval**) |
| cpp | int **set_signalRange**( const string& **newval**) |
| m | -(int) setSignalRange : (NSString*) **newval** |
| pas | function **set_signalRange**( **newval**: string): integer |
| vb | function **set_signalRange**( ByVal **newval** As String) As Integer |
| cs | int **set_signalRange**( string **newval**) |
| java | int **set_signalRange**( String **newval**) |
| py | def **set_signalRange**( **newval**) |
| cmd | YGenericSensor **target** **set_signalRange** **newval** |

**Parameters :**

**newval** a string corresponding to the electric signal range used by the sensor

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## generricsensor→set_unit() generricsensor→setUnit()[genericsensor setUnit: ] YGenericSensor

Changes the measuring unit for the measured value.

| | |
|---|---|
| js | function **set_unit**( **newval**) |
| nodejs | function **set_unit**( **newval**) |
| php | function **set_unit**( **$newval**) |
| cpp | int **set_unit**( const string& **newval**) |
| m | -(int) setUnit : (NSString*) **newval** |
| pas | function **set_unit**( **newval**: string): integer |
| vb | function **set_unit**( ByVal **newval** As String) As Integer |
| cs | int **set_unit**( string **newval**) |
| java | int **set_unit**( String **newval**) |
| py | def **set_unit**( **newval**) |
| cmd | YGenericSensor **target** **set_unit** **newval** |

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the measuring unit for the measured value

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→set_userData() genericsensor→setUserData()[genericsensor setUserData: ] YGenericSensor

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## genericsensor→set_valueRange() genericsensor→setValueRange()[genericsensor setValueRange: ]

**YGenericSensor**

Changes the physical value range measured by the sensor.

| | |
|---|---|
| js | function **set_valueRange**( **newval**) |
| nodejs | function **set_valueRange**( **newval**) |
| php | function **set_valueRange**( **$newval**) |
| cpp | int **set_valueRange**( const string& **newval**) |
| m | -(int) setValueRange : (NSString*) **newval** |
| pas | function **set_valueRange**( **newval**: string): integer |
| vb | function **set_valueRange**( ByVal **newval** As String) As Integer |
| cs | int **set_valueRange**( string **newval**) |
| java | int **set_valueRange**( String **newval**) |
| py | def **set_valueRange**( **newval**) |
| cmd | YGenericSensor **target** **set_valueRange** **newval** |

The range change may have a side effect on the display resolution, as it may be adapted automatically.

**Parameters :**

**newval** a string corresponding to the physical value range measured by the sensor

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## genericsensor→wait_async() YGenericSensor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

`js` function **wait_async**( **callback**, **context**)
`nodejs` function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.18. Gyroscope function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_gyro.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YGyro = yoctolib.YGyro; |
| php | require_once('yocto_gyro.php'); |
| cpp | #include "yocto_gyro.h" |
| m | #import "yocto_gyro.h" |
| pas | uses yocto_gyro; |
| vb | yocto_gyro.vb |
| cs | yocto_gyro.cs |
| java | import com.yoctopuce.YoctoAPI.YGyro; |
| py | from yocto_gyro import * |

## Global functions

**yFindGyro(func)**

Retrieves a gyroscope for a given identifier.

**yFirstGyro()**

Starts the enumeration of gyroscopes currently accessible.

## `YGyro` methods

**gyro→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**gyro→describe()**

Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**gyro→get_advertisedValue()**

Returns the current value of the gyroscope (no more than 6 characters).

**gyro→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**gyro→get_currentValue()**

Returns the current value of the angular velocity.

**gyro→get_errorMessage()**

Returns the error message of the latest error with the gyroscope.

**gyro→get_errorType()**

Returns the numerical error code of the latest error with the gyroscope.

**gyro→get_friendlyName()**

Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.

**gyro→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**gyro→get_functionId()**

Returns the hardware identifier of the gyroscope, without reference to the module.

**gyro→get_hardwareId()**

Returns the unique hardware identifier of the gyroscope in the form `SERIAL.FUNCTIONID`.

**gyro→get_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_highestValue()**

Returns the maximal value observed for the angular velocity since the device was started.

**gyro→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**gyro→get_logicalName()**

Returns the logical name of the gyroscope.

**gyro→get_lowestValue()**

Returns the minimal value observed for the angular velocity since the device was started.

**gyro→get_module()**

Gets the `YModule` object for the device on which the function is located.

**gyro→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**gyro→get_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionW()**

Returns the `w` component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionX()**

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionY()**

Returns the `y` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionZ()**

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**gyro→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**gyro→get_resolution()**

Returns the resolution of the measured values.

**gyro→get_roll()**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_unit()**

Returns the measuring unit for the angular velocity.

**gyro→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**gyro→get_xValue()**

Returns the angular velocity around the X axis of the device, as a floating point number.

**gyro→get_yValue()**

Returns the angular velocity around the Y axis of the device, as a floating point number.

**gyro→get_zValue()**

Returns the angular velocity around the Z axis of the device, as a floating point number.

**gyro→isOnline()**

Checks if the gyroscope is currently reachable, without raising any error.

**gyro→isOnline_async(callback, context)**

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

**gyro→load(msValidity)**

Preloads the gyroscope cache with a specified validity duration.

**gyro→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**gyro→load_async(msValidity, callback, context)**

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

**gyro→nextGyro()**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

**gyro→registerAnglesCallback(callback)**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

**gyro→registerQuaternionCallback(callback)**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

**gyro→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**gyro→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**gyro→set_highestValue(newval)**

Changes the recorded maximal value observed.

**gyro→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**gyro→set_logicalName(newval)**

Changes the logical name of the gyroscope.

**gyro→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**gyro→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**gyro→set_resolution(newval)**

Changes the resolution of the measured physical values.

**gyro→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**gyro→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YGyro.FindGyro() yFindGyro()yFindGyro() YGyro

Retrieves a gyroscope for a given identifier.

| | |
|---|---|
| js | function **yFindGyro**( **func**) |
| nodejs | function **FindGyro**( **func**) |
| php | function **yFindGyro**( **$func**) |
| cpp | YGyro* **yFindGyro**( string **func**) |
| m | +(YGyro*) **yFindGyro** : (NSString*) **func** |
| pas | function **yFindGyro**( **func**: string): TYGyro |
| vb | function **yFindGyro**( ByVal **func** As String) As YGyro |
| cs | YGyro **FindGyro**( string **func**) |
| java | YGyro **FindGyro**( String **func**) |
| py | def **FindGyro**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the gyroscope is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGyro.isOnline()` to test if the gyroscope is indeed online at a given time. In case of ambiguity when looking for a gyroscope by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the gyroscope

**Returns :**

a `YGyro` object allowing you to drive the gyroscope.

## YGyro.FirstGyro() yFirstGyro()yFirstGyro() — YGyro

Starts the enumeration of gyroscopes currently accessible.

| | |
|---|---|
| `js` | function **yFirstGyro**( ) |
| `nodejs` | function **FirstGyro**( ) |
| `php` | function **yFirstGyro**( ) |
| `cpp` | YGyro* **yFirstGyro**( ) |
| `m` | YGyro* **yFirstGyro**( ) |
| `pas` | function **yFirstGyro**( ): TYGyro |
| `vb` | function **yFirstGyro**( ) As YGyro |
| `cs` | YGyro **FirstGyro**( ) |
| `java` | YGyro **FirstGyro**( ) |
| `py` | def **FirstGyro**( ) |

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

**Returns :**

a pointer to a `YGyro` object, corresponding to the first gyro currently online, or a `null` pointer if there are none.

## gyro→calibrateFromPoints()[gyro calibrateFromPoints: ] YGyro

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**,<br>vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues**<br>: (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray,<br>**refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**,<br>List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**,<br>ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YGyro **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→describe()[gyro describe] YGyro

Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the gyroscope (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## gyro→get_advertisedValue() gyro→advertisedValue()[gyro advertisedValue] YGyro

Returns the current value of the gyroscope (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YGyro **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the gyroscope (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## gyro→get_currentRawValue() gyro→currentRawValue()[gyro currentRawValue] YGyro

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YGyro **target** **get_currentRawValue** |

**Returns :**
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## gyro→get_currentValue() gyro→currentValue()[gyro currentValue] YGyro

Returns the current value of the angular velocity.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YGyro **target** **get_currentValue** |

**Returns :**
a floating point number corresponding to the current value of the angular velocity

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## gyro→get_errorMessage() gyro→errorMessage()[gyro errorMessage] YGyro

Returns the error message of the latest error with the gyroscope.

```
js      function get_errorMessage( )
nodejs  function get_errorMessage( )
php     function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     function get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the gyroscope object

## gyro→get_errorType() gyro→errorType() YGyro

Returns the numerical error code of the latest error with the gyroscope.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the gyroscope object

## gyro→get_friendlyName() gyro→friendlyName()[gyro friendlyName] YGyro

Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the gyroscope if they are defined, otherwise the serial number of the module and the hardware identifier of the gyroscope (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the gyroscope using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## gyro→get_functionDescriptor() gyro→functionDescriptor()[gyro functionDescriptor] YGyro

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## gyro→get_functionId() gyro→functionId()[gyro functionId] YGyro

Returns the hardware identifier of the gyroscope, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the gyroscope (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## gyro→get_hardwareId() YGyro
## gyro→hardwareId()[gyro hardwareId]

Returns the unique hardware identifier of the gyroscope in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the gyroscope. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the gyroscope (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## gyro→get_heading() gyro→heading()[gyro heading] YGyro

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

| | |
|---|---|
| js | function **get_heading**( ) |
| nodejs | function **get_heading**( ) |
| php | function **get_heading**( ) |
| cpp | double **get_heading**( ) |
| m | -(double) heading |
| pas | function **get_heading**( ): double |
| vb | function **get_heading**( ) As Double |
| cs | double **get_heading**( ) |
| java | double **get_heading**( ) |
| py | def **get_heading**( ) |

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

**Returns :**
a floating-point number corresponding to heading in degrees, between 0 and 360.

## gyro→get_highestValue()<br>gyro→highestValue()[gyro highestValue] YGyro

Returns the maximal value observed for the angular velocity since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YGyro **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## gyro→get_logFrequency() gyro→logFrequency()[gyro logFrequency] YGyro

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YGyro **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## gyro→get_logicalName() gyro→logicalName()[gyro logicalName] YGyro

Returns the logical name of the gyroscope.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YGyro **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the gyroscope. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## gyro→get_lowestValue() gyro→lowestValue()[gyro lowestValue] YGyro

Returns the minimal value observed for the angular velocity since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YGyro **target** **get_lowestValue** |

**Returns :**

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## gyro→get_module() gyro→module()[gyro module] YGyro

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## gyro→get_module_async() gyro→module_async() YGyro

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## gyro→get_pitch() gyro→pitch()[gyro pitch] YGyro

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

| | |
|---|---|
| js | function **get_pitch**( ) |
| nodejs | function **get_pitch**( ) |
| php | function **get_pitch**( ) |
| cpp | double **get_pitch**( ) |
| m | -(double) pitch |
| pas | function **get_pitch**( ): double |
| vb | function **get_pitch**( ) As Double |
| cs | double **get_pitch**( ) |
| java | double **get_pitch**( ) |
| py | def **get_pitch**( ) |

The axis corresponding to the pitch angle can be mapped to any of the device `X`, `Y` or `Z` physical directions using methods of the class `YRefFrame`.

**Returns :**
a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

## gyro→get_quaternionW() gyro→quaternionW()[gyro quaternionW] YGyro

Returns the `w` component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

| | |
|---|---|
| js | function **get_quaternionW**( ) |
| nodejs | function **get_quaternionW**( ) |
| php | function **get_quaternionW**( ) |
| cpp | double **get_quaternionW**( ) |
| m | -(double) quaternionW |
| pas | function **get_quaternionW**( ): double |
| vb | function **get_quaternionW**( ) As Double |
| cs | double **get_quaternionW**( ) |
| java | double **get_quaternionW**( ) |
| py | def **get_quaternionW**( ) |

**Returns :**

a floating-point number corresponding to the `w` component of the quaternion.

## gyro→get_quaternionX() YGyro
## gyro→quaternionX()[gyro 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.

| | |
|---|---|
| js | function **get_quaternionX**( ) |
| nodejs | function **get_quaternionX**( ) |
| php | function **get_quaternionX**( ) |
| cpp | double **get_quaternionX**( ) |
| m | -(double) quaternionX |
| pas | function **get_quaternionX**( ): double |
| vb | function **get_quaternionX**( ) As Double |
| cs | double **get_quaternionX**( ) |
| java | double **get_quaternionX**( ) |
| py | def **get_quaternionX**( ) |

The X component is mostly correlated with rotations on the roll axis.

**Returns :**

a floating-point number corresponding to the X component of the quaternion.

## gyro→get_quaternionY() gyro→quaternionY()[gyro quaternionY] YGyro

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

| | |
|---|---|
| js | function **get_quaternionY**( ) |
| nodejs | function **get_quaternionY**( ) |
| php | function **get_quaternionY**( ) |
| cpp | double **get_quaternionY**( ) |
| m | -(double) quaternionY |
| pas | function **get_quaternionY**( ): double |
| vb | function **get_quaternionY**( ) As Double |
| cs | double **get_quaternionY**( ) |
| java | double **get_quaternionY**( ) |
| py | def **get_quaternionY**( ) |

The y component is mostly correlated with rotations on the pitch axis.

**Returns :**

a floating-point number corresponding to the y component of the quaternion.

## gyro→get_quaternionZ() gyro→quaternionZ()[gyro quaternionZ] YGyro

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

| | |
|---|---|
| js | function **get_quaternionZ**( ) |
| nodejs | function **get_quaternionZ**( ) |
| php | function **get_quaternionZ**( ) |
| cpp | double **get_quaternionZ**( ) |
| m | -(double) quaternionZ |
| pas | function **get_quaternionZ**( ): double |
| vb | function **get_quaternionZ**( ) As Double |
| cs | double **get_quaternionZ**( ) |
| java | double **get_quaternionZ**( ) |
| py | def **get_quaternionZ**( ) |

The x component is mostly correlated with changes of heading.

**Returns :**

a floating-point number corresponding to the z component of the quaternion.

## gyro→get_recordedData() gyro→recordedData()[gyro recordedData: ] YGyro

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime** : (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YGyro **target get_recordedData startTime endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## gyro→get_reportFrequency() gyro→reportFrequency()[gyro reportFrequency] YGyro

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YGyro **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## gyro→get_resolution() gyro→resolution()[gyro resolution] YGyro

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YGyro **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## gyro→get_roll() gyro→roll()[gyro roll] YGyro

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

| | |
|---|---|
| js | function **get_roll**( ) |
| nodejs | function **get_roll**( ) |
| php | function **get_roll**( ) |
| cpp | double **get_roll**( ) |
| m | -(double) roll |
| pas | function **get_roll**( ): double |
| vb | function **get_roll**( ) As Double |
| cs | double **get_roll**( ) |
| java | double **get_roll**( ) |
| py | def **get_roll**( ) |

The axis corresponding to the roll angle can be mapped to any of the device `X`, `Y` or `Z` physical directions using methods of the class `YRefFrame`.

**Returns :**
a floating-point number corresponding to roll angle in degrees, between -180 and +180.

## gyro→get_unit() gyro→unit()[gyro unit] YGyro

Returns the measuring unit for the angular velocity.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YGyro **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## gyro→get_userData() gyro→userData()[gyro userData] YGyro

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## gyro→get_xValue() gyro→xValue()[gyro xValue] YGyro

Returns the angular velocity around the X axis of the device, as a floating point number.

| | |
|---|---|
| js | function **get_xValue**( ) |
| nodejs | function **get_xValue**( ) |
| php | function **get_xValue**( ) |
| cpp | double **get_xValue**( ) |
| m | -(double) xValue |
| pas | function **get_xValue**( ): double |
| vb | function **get_xValue**( ) As Double |
| cs | double **get_xValue**( ) |
| java | double **get_xValue**( ) |
| py | def **get_xValue**( ) |

**Returns :**

a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns `Y_XVALUE_INVALID`.

## gyro→get_yValue() gyro→yValue()[gyro yValue] YGyro

Returns the angular velocity around the Y axis of the device, as a floating point number.

| | |
|---|---|
| js | function **get_yValue**( ) |
| nodejs | function **get_yValue**( ) |
| php | function **get_yValue**( ) |
| cpp | double **get_yValue**( ) |
| m | -(double) yValue |
| pas | function **get_yValue**( ): double |
| vb | function **get_yValue**( ) As Double |
| cs | double **get_yValue**( ) |
| java | double **get_yValue**( ) |
| py | def **get_yValue**( ) |

**Returns :**

a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

## gyro→get_zValue()<br>gyro→zValue()[gyro zValue] YGyro

Returns the angular velocity around the Z axis of the device, as a floating point number.

| | |
|---|---|
| js | function **get_zValue**( ) |
| nodejs | function **get_zValue**( ) |
| php | function **get_zValue**( ) |
| cpp | double **get_zValue**( ) |
| m | -(double) zValue |
| pas | function **get_zValue**( ): double |
| vb | function **get_zValue**( ) As Double |
| cs | double **get_zValue**( ) |
| java | double **get_zValue**( ) |
| py | def **get_zValue**( ) |

**Returns :**

a floating point number corresponding to the angular velocity around the Z axis of the device, as a floating point number

On failure, throws an exception or returns `Y_ZVALUE_INVALID`.

## gyro→isOnline()[gyro isOnline] YGyro

Checks if the gyroscope is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

**Returns :**

`true` if the gyroscope can be reached, and `false` otherwise

## gyro→isOnline_async() YGyro

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## gyro→load()[gyro load: ] YGyro

Preloads the gyroscope cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## gyro→loadCalibrationPoints()[gyro loadCalibrationPoints: ] YGyro

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| `js` | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| `nodejs` | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| `php` | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| `cpp` | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| `m` | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| `pas` | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| `vb` | procedure **loadCalibrationPoints**( ) |
| `cs` | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| `java` | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| `py` | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| `cmd` | YGyro **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→load_async() YGyro

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

js function **load_async**( **msValidity**, **callback**, **context**)
nodejs function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## gyro→nextGyro()[gyro nextGyro] YGyro

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

| | |
|---|---|
| js | function **nextGyro**( ) |
| nodejs | function **nextGyro**( ) |
| php | function **nextGyro**( ) |
| cpp | YGyro * **nextGyro**( ) |
| m | -(YGyro*) **nextGyro** |
| pas | function **nextGyro**( ): TYGyro |
| vb | function **nextGyro**( ) As YGyro |
| cs | YGyro **nextGyro**( ) |
| java | YGyro **nextGyro**( ) |
| py | def **nextGyro**( ) |

**Returns :**

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.

## gyro→registerAnglesCallback()[gyro registerAnglesCallback: ] YGyro

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

| | |
|---|---|
| js | function **registerAnglesCallback**( **callback**) |
| nodejs | function **registerAnglesCallback**( **callback**) |
| php | function **registerAnglesCallback**( **$callback**) |
| cpp | int **registerAnglesCallback**( YAnglesCallback **callback**) |
| m | -(int) **registerAnglesCallback** : (YAnglesCallback) **callback** |
| pas | function **registerAnglesCallback**( **callback**: TYAnglesCallback): LongInt |
| vb | function **registerAnglesCallback**( ) As Integer |
| cs | int **registerAnglesCallback**( YAnglesCallback **callback**) |
| java | int **registerAnglesCallback**( YAnglesCallback **callback**) |
| py | def **registerAnglesCallback**( **callback**) |

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGyro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

## gyro→registerQuaternionCallback()[gyro registerQuaternionCallback: ] YGyro

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

| | |
|---|---|
| js | function **registerQuaternionCallback**( **callback**) |
| nodejs | function **registerQuaternionCallback**( **callback**) |
| php | function **registerQuaternionCallback**( **$callback**) |
| cpp | int **registerQuaternionCallback**( YQuatCallback **callback**) |
| m | -(int) **registerQuaternionCallback** : (YQuatCallback) **callback** |
| pas | function **registerQuaternionCallback**( **callback**: TYQuatCallback): LongInt |
| vb | function **registerQuaternionCallback**( ) As Integer |
| cs | int **registerQuaternionCallback**( YQuatCallback **callback**) |
| java | int **registerQuaternionCallback**( YQuatCallback **callback**) |
| py | def **registerQuaternionCallback**( **callback**) |

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

## gyro→registerTimedReportCallback()[gyro registerTimedReportCallback: ] YGyro

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YGyroTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YGyroTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYGyroTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## gyro→registerValueCallback()[gyro registerValueCallback: ] YGyro

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YGyroValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YGyroValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYGyroValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## gyro→set_highestValue() gyro→setHighestValue()[gyro setHighestValue: ] YGyro

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YGyro **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→set_logFrequency() gyro→setLogFrequency()[gyro setLogFrequency: ] YGyro

Changes the datalogger recording frequency for this function.

```
js      function set_logFrequency( newval)
nodejs  function set_logFrequency( newval)
php     function set_logFrequency( $newval)
cpp     int set_logFrequency( const string& newval)
m       -(int) setLogFrequency : (NSString*) newval
pas     function set_logFrequency( newval: string): integer
vb      function set_logFrequency( ByVal newval As String) As Integer
cs      int set_logFrequency( string newval)
java    int set_logFrequency( String newval)
py      def set_logFrequency( newval)
cmd     YGyro target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→set_logicalName() gyro→setLogicalName()[gyro setLogicalName: ] YGyro

Changes the logical name of the gyroscope.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YGyro **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the gyroscope.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## gyro→set_lowestValue() gyro→setLowestValue()[gyro setLowestValue: ] YGyro

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YGyro **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→set_reportFrequency() gyro→setReportFrequency()[gyro setReportFrequency: ] YGyro

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YGyro **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→set_resolution() gyro→setResolution()[gyro setResolution: ] YGyro

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YGyro **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## gyro→set_userData() gyro→setUserData()[gyro setUserData: ] YGyro

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## gyro→wait_async() YGyro

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.19. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_hubport.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YHubPort = yoctolib.YHubPort; |
| php | require_once('yocto_hubport.php'); |
| cpp | #include "yocto_hubport.h" |
| m | #import "yocto_hubport.h" |
| pas | uses yocto_hubport; |
| vb | yocto_hubport.vb |
| cs | yocto_hubport.cs |
| java | import com.yoctopuce.YoctoAPI.YHubPort; |
| py | from yocto_hubport import * |

### Global functions

**yFindHubPort**(**func**)

Retrieves a Yocto-hub port for a given identifier.

**yFirstHubPort**()

Starts the enumeration of Yocto-hub ports currently accessible.

### `YHubPort` methods

**hubport→describe**()

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**hubport→get_advertisedValue**()

Returns the current value of the Yocto-hub port (no more than 6 characters).

**hubport→get_baudRate**()

Returns the current baud rate used by this Yocto-hub port, in kbps.

**hubport→get_enabled**()

Returns true if the Yocto-hub port is powered, false otherwise.

**hubport→get_errorMessage**()

Returns the error message of the latest error with the Yocto-hub port.

**hubport→get_errorType**()

Returns the numerical error code of the latest error with the Yocto-hub port.

**hubport→get_friendlyName**()

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

**hubport→get_functionDescriptor**()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**hubport→get_functionId**()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

**hubport→get_hardwareId**()

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

**hubport→get_logicalName**()

Returns the logical name of the Yocto-hub port.

**hubport→get_module()**

Gets the `YModule` object for the device on which the function is located.

**hubport→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**hubport→get_portState()**

Returns the current state of the Yocto-hub port.

**hubport→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**hubport→isOnline()**

Checks if the Yocto-hub port is currently reachable, without raising any error.

**hubport→isOnline_async(callback, context)**

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

**hubport→load(msValidity)**

Preloads the Yocto-hub port cache with a specified validity duration.

**hubport→load_async(msValidity, callback, context)**

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

**hubport→nextHubPort()**

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

**hubport→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**hubport→set_enabled(newval)**

Changes the activation of the Yocto-hub port.

**hubport→set_logicalName(newval)**

Changes the logical name of the Yocto-hub port.

**hubport→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**hubport→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YHubPort.FindHubPort() yFindHubPort()yFindHubPort() YHubPort

Retrieves a Yocto-hub port for a given identifier.

| | |
|---|---|
| js | function **yFindHubPort**( **func**) |
| nodejs | function **FindHubPort**( **func**) |
| php | function **yFindHubPort**( **$func**) |
| cpp | YHubPort* **yFindHubPort**( const string& **func**) |
| m | YHubPort* **yFindHubPort**( NSString* **func**) |
| pas | function **yFindHubPort**( **func**: string): TYHubPort |
| vb | function **yFindHubPort**( ByVal **func** As String) As YHubPort |
| cs | YHubPort **FindHubPort**( string **func**) |
| java | YHubPort **FindHubPort**( String **func**) |
| py | def **FindHubPort**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Yocto-hub port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHubPort.isOnline()` to test if the Yocto-hub port is indeed online at a given time. In case of ambiguity when looking for a Yocto-hub port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the Yocto-hub port

**Returns :**

a `YHubPort` object allowing you to drive the Yocto-hub port.

## YHubPort.FirstHubPort() yFirstHubPort()yFirstHubPort() YHubPort

Starts the enumeration of Yocto-hub ports currently accessible.

| | |
|---|---|
| js | function **yFirstHubPort**( ) |
| nodejs | function **FirstHubPort**( ) |
| php | function **yFirstHubPort**( ) |
| cpp | YHubPort* **yFirstHubPort**( ) |
| m | YHubPort* **yFirstHubPort**( ) |
| pas | function **yFirstHubPort**( ): TYHubPort |
| vb | function **yFirstHubPort**( ) As YHubPort |
| cs | YHubPort **FirstHubPort**( ) |
| java | YHubPort **FirstHubPort**( ) |
| py | def **FirstHubPort**( ) |

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

**Returns :**

a pointer to a `YHubPort` object, corresponding to the first Yocto-hub port currently online, or a `null` pointer if there are none.

## hubport→describe()[hubport describe] YHubPort

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| `js` | function **describe**( ) |
| `nodejs` | function **describe**( ) |
| `php` | function **describe**( ) |
| `cpp` | string **describe**( ) |
| `m` | -(NSString*) **describe** |
| `pas` | function **describe**( ): string |
| `vb` | function **describe**( ) As String |
| `cs` | string **describe**( ) |
| `java` | String **describe**( ) |
| `py` | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the Yocto-hub port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## hubport→get_advertisedValue() hubport→advertisedValue()[hubport advertisedValue] YHubPort

Returns the current value of the Yocto-hub port (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YHubPort **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the Yocto-hub port (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## hubport→get_baudRate() hubport→baudRate()[hubport baudRate] YHubPort

Returns the current baud rate used by this Yocto-hub port, in kbps.

| | |
|---|---|
| js | function **get_baudRate**( ) |
| nodejs | function **get_baudRate**( ) |
| php | function **get_baudRate**( ) |
| cpp | int **get_baudRate**( ) |
| m | -(int) baudRate |
| pas | function **get_baudRate**( ): LongInt |
| vb | function **get_baudRate**( ) As Integer |
| cs | int **get_baudRate**( ) |
| java | int **get_baudRate**( ) |
| py | def **get_baudRate**( ) |
| cmd | YHubPort **target** **get_baudRate** |

The default value is 1000 kbps, but a slower rate may be used if communication problems are encountered.

**Returns :**
an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns `Y_BAUDRATE_INVALID`.

## hubport→get_enabled() hubport→enabled()[hubport enabled]

**YHubPort**

Returns true if the Yocto-hub port is powered, false otherwise.

| | |
|---|---|
| js | function **get_enabled**( ) |
| nodejs | function **get_enabled**( ) |
| php | function **get_enabled**( ) |
| cpp | Y_ENABLED_enum **get_enabled**( ) |
| m | -(Y_ENABLED_enum) enabled |
| pas | function **get_enabled**( ): Integer |
| vb | function **get_enabled**( ) As Integer |
| cs | int **get_enabled**( ) |
| java | int **get_enabled**( ) |
| py | def **get_enabled**( ) |
| cmd | YHubPort **target** **get_enabled** |

**Returns :**

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the Yocto-hub port is powered, false otherwise

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

## hubport→get_errorMessage() hubport→errorMessage()[hubport errorMessage] YHubPort

Returns the error message of the latest error with the Yocto-hub port.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the Yocto-hub port object

## hubport→get_errorType() hubport→errorType() YHubPort

Returns the numerical error code of the latest error with the Yocto-hub port.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the Yocto-hub port object

## hubport→get_friendlyName() hubport→friendlyName()[hubport friendlyName] YHubPort

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the Yocto-hub port if they are defined, otherwise the serial number of the module and the hardware identifier of the Yocto-hub port (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the Yocto-hub port using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## hubport→get_functionDescriptor() hubport→functionDescriptor()[hubport functionDescriptor] YHubPort

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## hubport→get_functionId() hubport→functionId()[hubport functionId] YHubPort

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the Yocto-hub port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## hubport→get_hardwareId() hubport→hardwareId()[hubport hardwareId] YHubPort

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Yocto-hub port. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the Yocto-hub port (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## hubport→get_logicalName() hubport→logicalName()[hubport logicalName] YHubPort

Returns the logical name of the Yocto-hub port.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YHubPort **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the Yocto-hub port. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## hubport→get_module() hubport→module()[hubport module]

**YHubPort**

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## hubport→get_module_async() hubport→module_async() YHubPort

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## hubport→get_portState() hubport→portState()[hubport portState] YHubPort

Returns the current state of the Yocto-hub port.

| | |
|---|---|
| js | function **get_portState**( ) |
| nodejs | function **get_portState**( ) |
| php | function **get_portState**( ) |
| cpp | Y_PORTSTATE_enum **get_portState**( ) |
| m | -(Y_PORTSTATE_enum) portState |
| pas | function **get_portState**( ): Integer |
| vb | function **get_portState**( ) As Integer |
| cs | int **get_portState**( ) |
| java | int **get_portState**( ) |
| py | def **get_portState**( ) |
| cmd | YHubPort **target** **get_portState** |

**Returns :**

a value among Y_PORTSTATE_OFF, Y_PORTSTATE_OVRLD, Y_PORTSTATE_ON, Y_PORTSTATE_RUN and Y_PORTSTATE_PROG corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns Y_PORTSTATE_INVALID.

## hubport→get_userData() hubport→userData()[hubport userData] **YHubPort**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| `js` | function **get_userData**( ) |
| `nodejs` | function **get_userData**( ) |
| `php` | function **get_userData**( ) |
| `cpp` | void * **get_userData**( ) |
| `m` | -(void*) userData |
| `pas` | function **get_userData**( ): Tobject |
| `vb` | function **get_userData**( ) As Object |
| `cs` | object **get_userData**( ) |
| `java` | Object **get_userData**( ) |
| `py` | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## hubport→isOnline()[hubport isOnline] YHubPort

Checks if the Yocto-hub port is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Yocto-hub port.

**Returns :**
`true` if the Yocto-hub port can be reached, and `false` otherwise

## hubport→isOnline_async() YHubPort

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## hubport→load()[hubport load: ] YHubPort

Preloads the Yocto-hub port cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## hubport→load_async() YHubPort

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## hubport→nextHubPort()[hubport nextHubPort] YHubPort

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

| | |
|---|---|
| js | function **nextHubPort**( ) |
| nodejs | function **nextHubPort**( ) |
| php | function **nextHubPort**( ) |
| cpp | YHubPort * **nextHubPort**( ) |
| m | -(YHubPort*) **nextHubPort** |
| pas | function **nextHubPort**( ): TYHubPort |
| vb | function **nextHubPort**( ) As YHubPort |
| cs | YHubPort **nextHubPort**( ) |
| java | YHubPort **nextHubPort**( ) |
| py | def **nextHubPort**( ) |

**Returns :**

a pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a `null` pointer if there are no more Yocto-hub ports to enumerate.

## hubport→registerValueCallback()[hubport registerValueCallback: ] YHubPort

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YHubPortValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YHubPortValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYHubPortValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## hubport→set_enabled() hubport→setEnabled()[hubport setEnabled: ] YHubPort

Changes the activation of the Yocto-hub port.

| | |
|---|---|
| js | function **set_enabled**( **newval**) |
| nodejs | function **set_enabled**( **newval**) |
| php | function **set_enabled**( **$newval**) |
| cpp | int **set_enabled**( Y_ENABLED_enum **newval**) |
| m | -(int) setEnabled : (Y_ENABLED_enum) **newval** |
| pas | function **set_enabled**( **newval**: Integer): integer |
| vb | function **set_enabled**( ByVal **newval** As Integer) As Integer |
| cs | int **set_enabled**( int **newval**) |
| java | int **set_enabled**( int **newval**) |
| py | def **set_enabled**( **newval**) |
| cmd | YHubPort **target set_enabled newval** |

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

**Parameters :**

**newval** either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the activation of the Yocto-hub port

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## hubport→set_logicalName() hubport→setLogicalName()[hubport setLogicalName: ] YHubPort

Changes the logical name of the Yocto-hub port.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YHubPort **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the Yocto-hub port.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## hubport→set_userData() hubport→setUserData()[hubport setUserData: ] YHubPort

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## hubport→wait_async() YHubPort

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js function **wait_async**( **callback**, **context**)
nodejs function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.20. Humidity function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_humidity.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YHumidity = yoctolib.YHumidity; |
| php | require_once('yocto_humidity.php'); |
| cpp | #include "yocto_humidity.h" |
| m | #import "yocto_humidity.h" |
| pas | uses yocto_humidity; |
| vb | yocto_humidity.vb |
| cs | yocto_humidity.cs |
| java | import com.yoctopuce.YoctoAPI.YHumidity; |
| py | from yocto_humidity import * |

**Global functions**

**yFindHumidity(func)**

Retrieves a humidity sensor for a given identifier.

**yFirstHumidity()**

Starts the enumeration of humidity sensors currently accessible.

**`YHumidity` methods**

**humidity→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**humidity→describe()**

Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**humidity→get_advertisedValue()**

Returns the current value of the humidity sensor (no more than 6 characters).

**humidity→get_currentRawValue()**

Returns the unrounded and uncalibrated raw value returned by the sensor.

**humidity→get_currentValue()**

Returns the current measure for the humidity.

**humidity→get_errorMessage()**

Returns the error message of the latest error with the humidity sensor.

**humidity→get_errorType()**

Returns the numerical error code of the latest error with the humidity sensor.

**humidity→get_friendlyName()**

Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**humidity→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**humidity→get_functionId()**

Returns the hardware identifier of the humidity sensor, without reference to the module.

**humidity→get_hardwareId()**

Returns the unique hardware identifier of the humidity sensor in the form `SERIAL.FUNCTIONID`.

**humidity→get_highestValue()**

Returns the maximal value observed for the humidity.

**humidity→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**humidity→get_logicalName()**

Returns the logical name of the humidity sensor.

**humidity→get_lowestValue()**

Returns the minimal value observed for the humidity.

**humidity→get_module()**

Gets the `YModule` object for the device on which the function is located.

**humidity→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**humidity→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**humidity→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**humidity→get_resolution()**

Returns the resolution of the measured values.

**humidity→get_unit()**

Returns the measuring unit for the humidity.

**humidity→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**humidity→isOnline()**

Checks if the humidity sensor is currently reachable, without raising any error.

**humidity→isOnline_async(callback, context)**

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

**humidity→load(msValidity)**

Preloads the humidity sensor cache with a specified validity duration.

**humidity→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**humidity→load_async(msValidity, callback, context)**

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

**humidity→nextHumidity()**

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

**humidity→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**humidity→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**humidity→set_highestValue(newval)**

Changes the recorded maximal value observed for the humidity.

**humidity→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**humidity→set_logicalName(newval)**

Changes the logical name of the humidity sensor.

**humidity→set_lowestValue(newval)**

Changes the recorded minimal value observed for the humidity.

**humidity→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**humidity→set_resolution(newval)**

Changes the resolution of the measured physical values.

**humidity→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**humidity→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YHumidity.FindHumidity() yFindHumidity()yFindHumidity()

YHumidity

Retrieves a humidity sensor for a given identifier.

| | |
|---|---|
| js | function **yFindHumidity**( **func**) |
| nodejs | function **FindHumidity**( **func**) |
| php | function **yFindHumidity**( **$func**) |
| cpp | YHumidity* **yFindHumidity**( const string& **func**) |
| m | YHumidity* **yFindHumidity**( NSString* **func**) |
| pas | function **yFindHumidity**( **func**: string): TYHumidity |
| vb | function **yFindHumidity**( ByVal **func** As String) As YHumidity |
| cs | YHumidity **FindHumidity**( string **func**) |
| java | YHumidity **FindHumidity**( String **func**) |
| py | def **FindHumidity**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the humidity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHumidity.isOnline()` to test if the humidity sensor is indeed online at a given time. In case of ambiguity when looking for a humidity sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the humidity sensor

**Returns :**

a `YHumidity` object allowing you to drive the humidity sensor.

## YHumidity.FirstHumidity() yFirstHumidity()yFirstHumidity() YHumidity

Starts the enumeration of humidity sensors currently accessible.

| | |
|---|---|
| js | function **yFirstHumidity**( ) |
| nodejs | function **FirstHumidity**( ) |
| php | function **yFirstHumidity**( ) |
| cpp | YHumidity* **yFirstHumidity**( ) |
| m | YHumidity* **yFirstHumidity**( ) |
| pas | function **yFirstHumidity**( ): TYHumidity |
| vb | function **yFirstHumidity**( ) As YHumidity |
| cs | YHumidity **FirstHumidity**( ) |
| java | YHumidity **FirstHumidity**( ) |
| py | def **FirstHumidity**( ) |

Use the method `YHumidity.nextHumidity()` to iterate on next humidity sensors.

**Returns :**

a pointer to a `YHumidity` object, corresponding to the first humidity sensor currently online, or a `null` pointer if there are none.

## humidity→calibrateFromPoints()[humidity calibrateFromPoints: ] YHumidity

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YHumidity **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→describe()[humidity describe] YHumidity

Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the humidity sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## humidity→get_advertisedValue() humidity→advertisedValue()[humidity advertisedValue] YHumidity

Returns the current value of the humidity sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YHumidity **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the humidity sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## humidity→get_currentRawValue() humidity→currentRawValue()[humidity currentRawValue] YHumidity

Returns the unrounded and uncalibrated raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YHumidity **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## humidity→get_currentValue() humidity→currentValue()[humidity currentValue] YHumidity

Returns the current measure for the humidity.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YHumidity **target** **get_currentValue** |

**Returns :**
a floating point number corresponding to the current measure for the humidity

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## humidity→get_errorMessage() humidity→errorMessage()[humidity errorMessage] YHumidity

Returns the error message of the latest error with the humidity sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the humidity sensor object

## humidity→get_errorType() humidity→errorType() — YHumidity

Returns the numerical error code of the latest error with the humidity sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the humidity sensor object

## humidity→get_friendlyName() humidity→friendlyName()[humidity friendlyName]

YHumidity

Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the humidity sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the humidity sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the humidity sensor using logical names (ex: `MyCustomName.relay1`)
On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## humidity→get_functionDescriptor() humidity→functionDescriptor()[humidity functionDescriptor] YHumidity

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## humidity→get_functionId() humidity→functionId()[humidity functionId] YHumidity

Returns the hardware identifier of the humidity sensor, without reference to the module.

| | |
|---|---|
| `js` | function **get_functionId**( ) |
| `nodejs` | function **get_functionId**( ) |
| `php` | function **get_functionId**( ) |
| `cpp` | string **get_functionId**( ) |
| `m` | -(NSString*) functionId |
| `vb` | function **get_functionId**( ) As String |
| `cs` | string **get_functionId**( ) |
| `java` | String **get_functionId**( ) |
| `py` | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the humidity sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## humidity→get_hardwareId() humidity→hardwareId()[humidity hardwareId] YHumidity

Returns the unique hardware identifier of the humidity sensor in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the humidity sensor. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the humidity sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## humidity→get_highestValue() humidity→highestValue()[humidity highestValue] YHumidity

Returns the maximal value observed for the humidity.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YHumidity **target** **get_highestValue** |

**Returns :**
a floating point number corresponding to the maximal value observed for the humidity

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## humidity→get_logFrequency() humidity→logFrequency()[humidity logFrequency] YHumidity

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YHumidity **target** **get_logFrequency** |

**Returns :**
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## humidity→get_logicalName() humidity→logicalName()[humidity logicalName] YHumidity

Returns the logical name of the humidity sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YHumidity **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the humidity sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## humidity→get_lowestValue() humidity→lowestValue()[humidity lowestValue] YHumidity

Returns the minimal value observed for the humidity.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YHumidity **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the humidity

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## humidity→get_module() humidity→module()[humidity module] YHumidity

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## humidity→get_module_async() humidity→module_async() YHumidity

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## humidity→get_recordedData() humidity→recordedData()[humidity recordedData: ] YHumidity

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YHumidity **target get_recordedData startTime endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## humidity→get_reportFrequency() humidity→reportFrequency()[humidity reportFrequency] YHumidity

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YHumidity **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## humidity→get_resolution() humidity→resolution()[humidity resolution] YHumidity

Returns the resolution of the measured values.

| | |
|---|---|
| `js` | function **get_resolution**( ) |
| `nodejs` | function **get_resolution**( ) |
| `php` | function **get_resolution**( ) |
| `cpp` | double **get_resolution**( ) |
| `m` | -(double) resolution |
| `pas` | function **get_resolution**( ): double |
| `vb` | function **get_resolution**( ) As Double |
| `cs` | double **get_resolution**( ) |
| `java` | double **get_resolution**( ) |
| `py` | def **get_resolution**( ) |
| `cmd` | YHumidity **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## humidity→get_unit() humidity→unit()[humidity unit] YHumidity

Returns the measuring unit for the humidity.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YHumidity **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## humidity→get_userData() humidity→userData()[humidity userData] YHumidity

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## humidity→isOnline()[humidity isOnline] YHumidity

Checks if the humidity sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the humidity sensor.

**Returns :**
`true` if the humidity sensor can be reached, and `false` otherwise

## humidity→isOnline_async() YHumidity

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## humidity→load()[humidity load: ] YHumidity

Preloads the humidity sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## humidity→loadCalibrationPoints()[humidity loadCalibrationPoints: ] YHumidity

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**,<br>vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues**<br>: (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray,<br>var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**,<br>List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**,<br>ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YHumidity **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→load_async() YHumidity

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## humidity→nextHumidity()[humidity nextHumidity] YHumidity

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

| | |
|---|---|
| js | function **nextHumidity**( ) |
| nodejs | function **nextHumidity**( ) |
| php | function **nextHumidity**( ) |
| cpp | YHumidity * **nextHumidity**( ) |
| m | -(YHumidity*) **nextHumidity** |
| pas | function **nextHumidity**( ): TYHumidity |
| vb | function **nextHumidity**( ) As YHumidity |
| cs | YHumidity **nextHumidity**( ) |
| java | YHumidity **nextHumidity**( ) |
| py | def **nextHumidity**( ) |

**Returns :**

a pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a `null` pointer if there are no more humidity sensors to enumerate.

## humidity→registerTimedReportCallback()[humidity registerTimedReportCallback: ] YHumidity

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YHumidityTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YHumidityTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYHumidityTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## humidity→registerValueCallback()[humidity registerValueCallback: ] YHumidity

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YHumidityValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YHumidityValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYHumidityValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## humidity→set_highestValue() humidity→setHighestValue()[humidity setHighestValue: ] YHumidity

Changes the recorded maximal value observed for the humidity.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YHumidity **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the humidity

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→set_logFrequency() humidity→setLogFrequency()[humidity setLogFrequency: ] YHumidity

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YHumidity **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→set_logicalName() humidity→setLogicalName()[humidity setLogicalName: ] YHumidity

Changes the logical name of the humidity sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YHumidity **target set_logicalName newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the humidity sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## humidity→set_lowestValue() humidity→setLowestValue()[humidity setLowestValue: ] YHumidity

Changes the recorded minimal value observed for the humidity.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YHumidity **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the humidity

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→set_reportFrequency() humidity→setReportFrequency()[humidity setReportFrequency: ] YHumidity

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YHumidity **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→set_resolution() humidity→setResolution()[humidity setResolution: ] YHumidity

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YHumidity **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→set_userData() humidity→setUserData()[humidity setUserData: ] YHumidity

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## humidity→wait_async() YHumidity

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

`js` function **wait_async**( **callback**, **context**)
`nodejs` function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.21. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_led.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YLed = yoctolib.YLed; |
| php | require_once('yocto_led.php'); |
| cpp | #include "yocto_led.h" |
| m | #import "yocto_led.h" |
| pas | uses yocto_led; |
| vb | yocto_led.vb |
| cs | yocto_led.cs |
| java | import com.yoctopuce.YoctoAPI.YLed; |
| py | from yocto_led import * |

| Global functions |
|---|
| **yFindLed**(**func**)<br>Retrieves a led for a given identifier. |
| **yFirstLed**()<br>Starts the enumeration of leds currently accessible. |
| **YLed methods** |
| **led→describe**()<br>Returns a short text that describes unambiguously the instance of the led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **led→get_advertisedValue**()<br>Returns the current value of the led (no more than 6 characters). |
| **led→get_blinking**()<br>Returns the current led signaling mode. |
| **led→get_errorMessage**()<br>Returns the error message of the latest error with the led. |
| **led→get_errorType**()<br>Returns the numerical error code of the latest error with the led. |
| **led→get_friendlyName**()<br>Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`. |
| **led→get_functionDescriptor**()<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **led→get_functionId**()<br>Returns the hardware identifier of the led, without reference to the module. |
| **led→get_hardwareId**()<br>Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`. |
| **led→get_logicalName**()<br>Returns the logical name of the led. |
| **led→get_luminosity**()<br>Returns the current led intensity (in per cent). |
| **led→get_module**() |

Gets the `YModule` object for the device on which the function is located.

**led→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**led→get_power()**

Returns the current led state.

**led→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**led→isOnline()**

Checks if the led is currently reachable, without raising any error.

**led→isOnline_async(callback, context)**

Checks if the led is currently reachable, without raising any error (asynchronous version).

**led→load(msValidity)**

Preloads the led cache with a specified validity duration.

**led→load_async(msValidity, callback, context)**

Preloads the led cache with a specified validity duration (asynchronous version).

**led→nextLed()**

Continues the enumeration of leds started using `yFirstLed()`.

**led→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**led→set_blinking(newval)**

Changes the current led signaling mode.

**led→set_logicalName(newval)**

Changes the logical name of the led.

**led→set_luminosity(newval)**

Changes the current led intensity (in per cent).

**led→set_power(newval)**

Changes the state of the led.

**led→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**led→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YLed.FindLed() yFindLed()yFindLed() YLed

Retrieves a led for a given identifier.

| | |
|---|---|
| js | function **yFindLed**( **func**) |
| nodejs | function **FindLed**( **func**) |
| php | function **yFindLed**( **$func**) |
| cpp | YLed* **yFindLed**( const string& **func**) |
| m | YLed* **yFindLed**( NSString* **func**) |
| pas | function **yFindLed**( **func**: string): TYLed |
| vb | function **yFindLed**( ByVal **func** As String) As YLed |
| cs | YLed **FindLed**( string **func**) |
| java | YLed **FindLed**( String **func**) |
| py | def **FindLed**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the led is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLed.isOnline()` to test if the led is indeed online at a given time. In case of ambiguity when looking for a led by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the led

**Returns :**

a `YLed` object allowing you to drive the led.

## YLed.FirstLed() yFirstLed()yFirstLed() YLed

Starts the enumeration of leds currently accessible.

| | |
|---|---|
| js | function **yFirstLed**( ) |
| nodejs | function **FirstLed**( ) |
| php | function **yFirstLed**( ) |
| cpp | YLed* **yFirstLed**( ) |
| m | YLed* **yFirstLed**( ) |
| pas | function **yFirstLed**( ): TYLed |
| vb | function **yFirstLed**( ) As YLed |
| cs | YLed **FirstLed**( ) |
| java | YLed **FirstLed**( ) |
| py | def **FirstLed**( ) |

Use the method `YLed.nextLed()` to iterate on next leds.

**Returns :**

a pointer to a `YLed` object, corresponding to the first led currently online, or a `null` pointer if there are none.

## led→describe()[led describe] YLed

Returns a short text that describes unambiguously the instance of the led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## led→get_advertisedValue() led→advertisedValue()[led advertisedValue] YLed

Returns the current value of the led (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YLed **target** **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()<br>led→blinking()[led blinking] — YLed

Returns the current led signaling mode.

| | |
|---|---|
| js | function **get_blinking**( ) |
| nodejs | function **get_blinking**( ) |
| php | function **get_blinking**( ) |
| cpp | Y_BLINKING_enum **get_blinking**( ) |
| m | -(Y_BLINKING_enum) blinking |
| pas | function **get_blinking**( ): Integer |
| vb | function **get_blinking**( ) As Integer |
| cs | int **get_blinking**( ) |
| java | int **get_blinking**( ) |
| py | def **get_blinking**( ) |
| cmd | YLed **target** **get_blinking** |

**Returns :**

a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

On failure, throws an exception or returns `Y_BLINKING_INVALID`.

## led→get_errorMessage() led→errorMessage()[led errorMessage] YLed

Returns the error message of the latest error with the led.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the led object

## led→get_errorType() led→errorType() YLed

Returns the numerical error code of the latest error with the led.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the led object

## led→get_friendlyName() led→friendlyName()[led friendlyName] YLed

Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the led if they are defined, otherwise the serial number of the module and the hardware identifier of the led (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the led using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## led→get_functionDescriptor() led→functionDescriptor()[led functionDescriptor] YLed

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## led→get_functionId() YLed
## led→functionId()[led functionId]

Returns the hardware identifier of the led, without reference to the module.

| | |
|---|---|
| `js` | function **get_functionId**( ) |
| `nodejs` | function **get_functionId**( ) |
| `php` | function **get_functionId**( ) |
| `cpp` | string **get_functionId**( ) |
| `m` | -(NSString*) functionId |
| `vb` | function **get_functionId**( ) As String |
| `cs` | string **get_functionId**( ) |
| `java` | String **get_functionId**( ) |
| `py` | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the led (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## led→get_hardwareId() YLed
## led→hardwareId()[led hardwareId]

Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the led. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the led (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## led→get_logicalName() YLed
## led→logicalName()[led logicalName]

Returns the logical name of the led.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YLed **target** **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 luminosity]

Returns the current led intensity (in per cent).

| | |
|---|---|
| js | function **get_luminosity**( ) |
| nodejs | function **get_luminosity**( ) |
| php | function **get_luminosity**( ) |
| cpp | int **get_luminosity**( ) |
| m | -(int) luminosity |
| pas | function **get_luminosity**( ): LongInt |
| vb | function **get_luminosity**( ) As Integer |
| cs | int **get_luminosity**( ) |
| java | int **get_luminosity**( ) |
| py | def **get_luminosity**( ) |
| cmd | YLed **target** **get_luminosity** |

**Returns :**
an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

## led→get_module() led→module()[led module] YLed

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## led→get_module_async() led→module_async() YLed

Gets the YModule object for the device on which the function is located (asynchronous version).

| js | function **get_module_async**( **callback**, **context**) |
|---|---|
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## led→get_power() led→power()[led power]

**YLed**

Returns the current led state.

| | |
|---|---|
| js | function **get_power**( ) |
| nodejs | function **get_power**( ) |
| php | function **get_power**( ) |
| cpp | Y_POWER_enum **get_power**( ) |
| m | -(Y_POWER_enum) power |
| pas | function **get_power**( ): Integer |
| vb | function **get_power**( ) As Integer |
| cs | int **get_power**( ) |
| java | int **get_power**( ) |
| py | def **get_power**( ) |
| cmd | YLed **target** **get_power** |

**Returns :**

either `Y_POWER_OFF` or `Y_POWER_ON`, according to the current led state

On failure, throws an exception or returns `Y_POWER_INVALID`.

## led→get_userData() led→userData()[led userData] YLed

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## led→isOnline()[led isOnline] YLed

Checks if the led is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the led.

**Returns :**
`true` if the led can be reached, and `false` otherwise

## led→isOnline_async() YLed

Checks if the led is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## led→load()[led load: ] YLed

Preloads the led cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## led→load_async() YLed

Preloads the led cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## led→nextLed()[led nextLed] YLed

Continues the enumeration of leds started using `yFirstLed()`.

| | |
|---|---|
| js | function **nextLed**( ) |
| nodejs | function **nextLed**( ) |
| php | function **nextLed**( ) |
| cpp | YLed * **nextLed**( ) |
| m | -(YLed*) **nextLed** |
| pas | function **nextLed**( ): TYLed |
| vb | function **nextLed**( ) As YLed |
| cs | YLed **nextLed**( ) |
| java | YLed **nextLed**( ) |
| py | def **nextLed**( ) |

**Returns :**

a pointer to a `YLed` object, corresponding to a led currently online, or a `null` pointer if there are no more leds to enumerate.

## led→registerValueCallback()[led registerValueCallback: ] YLed

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YLedValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YLedValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYLedValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## led→set_blinking() YLed
## led→setBlinking()[led setBlinking: ]

Changes the current led signaling mode.

| | |
|---|---|
| js | function **set_blinking**( **newval**) |
| nodejs | function **set_blinking**( **newval**) |
| php | function **set_blinking**( **$newval**) |
| cpp | int **set_blinking**( Y_BLINKING_enum **newval**) |
| m | -(int) setBlinking : (Y_BLINKING_enum) **newval** |
| pas | function **set_blinking**( **newval**: Integer): integer |
| vb | function **set_blinking**( ByVal **newval** As Integer) As Integer |
| cs | int **set_blinking**( int **newval**) |
| java | int **set_blinking**( int **newval**) |
| py | def **set_blinking**( **newval**) |
| cmd | YLed **target** **set_blinking** **newval** |

**Parameters :**

**newval** a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## led→set_logicalName() led→setLogicalName()[led setLogicalName: ] YLed

Changes the logical name of the led.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YLed **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the led.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## led→set_luminosity() led→setLuminosity()[led setLuminosity: ] YLed

Changes the current led intensity (in per cent).

| | |
|---|---|
| js | function **set_luminosity**( **newval**) |
| nodejs | function **set_luminosity**( **newval**) |
| php | function **set_luminosity**( **$newval**) |
| cpp | int **set_luminosity**( int **newval**) |
| m | -(int) setLuminosity : (int) **newval** |
| pas | function **set_luminosity**( **newval**: LongInt): integer |
| vb | function **set_luminosity**( ByVal **newval** As Integer) As Integer |
| cs | int **set_luminosity**( int **newval**) |
| java | int **set_luminosity**( int **newval**) |
| py | def **set_luminosity**( **newval**) |
| cmd | YLed **target set_luminosity newval** |

**Parameters :**

**newval** an integer corresponding to the current led intensity (in per cent)

**Returns :**

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## led→set_power() led→setPower()[led setPower: ] YLed

Changes the state of the led.

| | |
|---|---|
| js | function **set_power**( **newval**) |
| nodejs | function **set_power**( **newval**) |
| php | function **set_power**( **$newval**) |
| cpp | int **set_power**( Y_POWER_enum **newval**) |
| m | -(int) setPower : (Y_POWER_enum) **newval** |
| pas | function **set_power**( **newval**: Integer): integer |
| vb | function **set_power**( ByVal **newval** As Integer) As Integer |
| cs | int **set_power**( int **newval**) |
| java | int **set_power**( int **newval**) |
| py | def **set_power**( **newval**) |
| cmd | YLed **target** **set_power** **newval** |

**Parameters :**

**newval** either `Y_POWER_OFF` or `Y_POWER_ON`, according to the state of the led

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## led→set_userData() led→setUserData()[led setUserData: ] YLed

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## led→wait_async() YLed

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.22. LightSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_lightsensor.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YLightSensor = yoctolib.YLightSensor; |
| php | require_once('yocto_lightsensor.php'); |
| cpp | #include "yocto_lightsensor.h" |
| m | #import "yocto_lightsensor.h" |
| pas | uses yocto_lightsensor; |
| vb | yocto_lightsensor.vb |
| cs | yocto_lightsensor.cs |
| java | import com.yoctopuce.YoctoAPI.YLightSensor; |
| py | from yocto_lightsensor import * |

**Global functions**

**yFindLightSensor(func)**

Retrieves a light sensor for a given identifier.

**yFirstLightSensor()**

Starts the enumeration of light sensors currently accessible.

**`YLightSensor` methods**

**lightsensor→calibrate(calibratedVal)**

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

**lightsensor→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**lightsensor→describe()**

Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**lightsensor→get_advertisedValue()**

Returns the current value of the light sensor (no more than 6 characters).

**lightsensor→get_currentRawValue()**

Returns the unrounded and uncalibrated raw value returned by the sensor.

**lightsensor→get_currentValue()**

Returns the current measure for the ambiant light.

**lightsensor→get_errorMessage()**

Returns the error message of the latest error with the light sensor.

**lightsensor→get_errorType()**

Returns the numerical error code of the latest error with the light sensor.

**lightsensor→get_friendlyName()**

Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**lightsensor→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**lightsensor→get_functionId()**

Returns the hardware identifier of the light sensor, without reference to the module.

**lightsensor→get_hardwareId()**

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

**lightsensor→get_highestValue()**

Returns the maximal value observed for the ambiant light.

**lightsensor→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**lightsensor→get_logicalName()**

Returns the logical name of the light sensor.

**lightsensor→get_lowestValue()**

Returns the minimal value observed for the ambiant light.

**lightsensor→get_module()**

Gets the `YModule` object for the device on which the function is located.

**lightsensor→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**lightsensor→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**lightsensor→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**lightsensor→get_resolution()**

Returns the resolution of the measured values.

**lightsensor→get_unit()**

Returns the measuring unit for the ambiant light.

**lightsensor→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**lightsensor→isOnline()**

Checks if the light sensor is currently reachable, without raising any error.

**lightsensor→isOnline_async(callback, context)**

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

**lightsensor→load(msValidity)**

Preloads the light sensor cache with a specified validity duration.

**lightsensor→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**lightsensor→load_async(msValidity, callback, context)**

Preloads the light sensor cache with a specified validity duration (asynchronous version).

**lightsensor→nextLightSensor()**

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

**lightsensor→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**lightsensor→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**lightsensor→set_highestValue(newval)**

Changes the recorded maximal value observed for the ambiant light.

**lightsensor→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**lightsensor→set_logicalName(newval)**

Changes the logical name of the light sensor.

**lightsensor→set_lowestValue(newval)**

Changes the recorded minimal value observed for the ambiant light.

**lightsensor→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**lightsensor→set_resolution(newval)**

Changes the resolution of the measured physical values.

**lightsensor→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**lightsensor→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YLightSensor.FindLightSensor() yFindLightSensor()yFindLightSensor() YLightSensor

Retrieves a light sensor for a given identifier.

| | |
|---|---|
| js | function **yFindLightSensor**( **func**) |
| nodejs | function **FindLightSensor**( **func**) |
| php | function **yFindLightSensor**( **$func**) |
| cpp | YLightSensor* **yFindLightSensor**( const string& **func**) |
| m | YLightSensor* **yFindLightSensor**( NSString* **func**) |
| pas | function **yFindLightSensor**( **func**: string): TYLightSensor |
| vb | function **yFindLightSensor**( ByVal **func** As String) As YLightSensor |
| cs | YLightSensor **FindLightSensor**( string **func**) |
| java | YLightSensor **FindLightSensor**( String **func**) |
| py | def **FindLightSensor**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the light sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLightSensor.isOnline()` to test if the light sensor is indeed online at a given time. In case of ambiguity when looking for a light sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the light sensor

**Returns :**

a `YLightSensor` object allowing you to drive the light sensor.

## YLightSensor.FirstLightSensor() yFirstLightSensor()yFirstLightSensor()

**YLightSensor**

Starts the enumeration of light sensors currently accessible.

| | |
|---|---|
| js | function **yFirstLightSensor**( ) |
| nodejs | function **FirstLightSensor**( ) |
| php | function **yFirstLightSensor**( ) |
| cpp | YLightSensor* **yFirstLightSensor**( ) |
| m | YLightSensor* **yFirstLightSensor**( ) |
| pas | function **yFirstLightSensor**( ): TYLightSensor |
| vb | function **yFirstLightSensor**( ) As YLightSensor |
| cs | YLightSensor **FirstLightSensor**( ) |
| java | YLightSensor **FirstLightSensor**( ) |
| py | def **FirstLightSensor**( ) |

Use the method `YLightSensor.nextLightSensor()` to iterate on next light sensors.

**Returns :**

a pointer to a `YLightSensor` object, corresponding to the first light sensor currently online, or a `null` pointer if there are none.

## lightsensor→calibrate()[lightsensor calibrate: ] YLightSensor

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

| | |
|---|---|
| js | function **calibrate**( **calibratedVal**) |
| nodejs | function **calibrate**( **calibratedVal**) |
| php | function **calibrate**( **$calibratedVal**) |
| cpp | int **calibrate**( double **calibratedVal**) |
| m | -(int) **calibrate** : (double) **calibratedVal** |
| pas | function **calibrate**( **calibratedVal**: double): integer |
| vb | function **calibrate**( ByVal **calibratedVal** As Double) As Integer |
| cs | int **calibrate**( double **calibratedVal**) |
| java | int **calibrate**( double **calibratedVal**) |
| py | def **calibrate**( **calibratedVal**) |
| cmd | YLightSensor **target** **calibrate** **calibratedVal** |

**Parameters :**

**calibratedVal** the desired target value.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→calibrateFromPoints()[lightsensor calibrateFromPoints: ] YLightSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YLightSensor **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→describe()[lightsensor describe] YLightSensor

Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the light sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## lightsensor→get_advertisedValue() lightsensor→advertisedValue()[lightsensor advertisedValue] YLightSensor

Returns the current value of the light sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YLightSensor **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the light sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## lightsensor→get_currentRawValue() lightsensor→currentRawValue()[lightsensor currentRawValue] YLightSensor

Returns the unrounded and uncalibrated raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YLightSensor **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## lightsensor→get_currentValue() lightsensor→currentValue()[lightsensor currentValue] YLightSensor

Returns the current measure for the ambiant light.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YLightSensor **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current measure for the ambiant light

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

### lightsensor→get_errorMessage() lightsensor→errorMessage()[lightsensor errorMessage] YLightSensor

Returns the error message of the latest error with the light sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the light sensor object

## lightsensor→get_errorType() lightsensor→errorType() YLightSensor

Returns the numerical error code of the latest error with the light sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the light sensor object

## lightsensor→get_friendlyName() lightsensor→friendlyName()[lightsensor friendlyName] YLightSensor

Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the light sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the light sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the light sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## lightsensor→get_functionDescriptor() lightsensor→functionDescriptor()[lightsensor functionDescriptor] YLightSensor

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## lightsensor→get_functionId() lightsensor→functionId()[lightsensor functionId] YLightSensor

Returns the hardware identifier of the light sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the light sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## lightsensor→get_hardwareId() lightsensor→hardwareId()[lightsensor hardwareId] YLightSensor

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the light sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the light sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## lightsensor→get_highestValue() lightsensor→highestValue()[lightsensor highestValue] YLightSensor

Returns the maximal value observed for the ambiant light.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YLightSensor **target** **get_highestValue** |

**Returns :**
a floating point number corresponding to the maximal value observed for the ambiant light

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## lightsensor→get_logFrequency() lightsensor→logFrequency()[lightsensor logFrequency] YLightSensor

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YLightSensor **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## lightsensor→get_logicalName() lightsensor→logicalName()[lightsensor logicalName] YLightSensor

Returns the logical name of the light sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YLightSensor **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the light sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## lightsensor→get_lowestValue() lightsensor→lowestValue()[lightsensor lowestValue] YLightSensor

Returns the minimal value observed for the ambiant light.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YLightSensor **target** **get_lowestValue** |

**Returns :**

a floating point number corresponding to the minimal value observed for the ambiant light

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## lightsensor→get_module() lightsensor→module()[lightsensor module] YLightSensor

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## lightsensor→get_module_async() lightsensor→module_async() YLightSensor

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| `js` | function **get_module_async**( **callback**, **context**) |
| `nodejs` | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## lightsensor→get_recordedData() lightsensor→recordedData()[lightsensor recordedData: ] YLightSensor

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YLightSensor **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## lightsensor→get_reportFrequency() lightsensor→reportFrequency()[lightsensor reportFrequency] YLightSensor

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YLightSensor **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## lightsensor→get_resolution() lightsensor→resolution()[lightsensor resolution] YLightSensor

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YLightSensor **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## lightsensor→get_unit() lightsensor→unit()[lightsensor unit] YLightSensor

Returns the measuring unit for the ambiant light.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YLightSensor **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the ambiant light

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## lightsensor→get_userData() lightsensor→userData()[lightsensor userData] YLightSensor

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## lightsensor→isOnline()[lightsensor isOnline] YLightSensor

Checks if the light sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

**Returns :**

`true` if the light sensor can be reached, and `false` otherwise

## lightsensor→isOnline_async() YLightSensor

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## lightsensor→load()[lightsensor load: ] YLightSensor

Preloads the light sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## lightsensor→loadCalibrationPoints()[lightsensor loadCalibrationPoints: ] YLightSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
js      function loadCalibrationPoints( rawValues, refValues)
nodejs  function loadCalibrationPoints( rawValues, refValues)
php     function loadCalibrationPoints( &$rawValues, &$refValues)
cpp     int loadCalibrationPoints( vector<double>& rawValues,
                                   vector<double>& refValues)
m       -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                                     : (NSMutableArray*) refValues
pas     function loadCalibrationPoints( var rawValues: TDoubleArray,
                                        var refValues: TDoubleArray): LongInt
vb      procedure loadCalibrationPoints( )
cs      int loadCalibrationPoints( List<double> rawValues,
                                   List<double> refValues)
java    int loadCalibrationPoints( ArrayList<Double> rawValues,
                                   ArrayList<Double> refValues)
py      def loadCalibrationPoints( rawValues, refValues)
cmd     YLightSensor target loadCalibrationPoints rawValues refValues
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→load_async() YLightSensor

Preloads the light sensor cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## lightsensor→nextLightSensor()[lightsensor nextLightSensor] YLightSensor

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

| | |
|---|---|
| js | function **nextLightSensor**( ) |
| nodejs | function **nextLightSensor**( ) |
| php | function **nextLightSensor**( ) |
| cpp | YLightSensor * **nextLightSensor**( ) |
| m | -(YLightSensor*) **nextLightSensor** |
| pas | function **nextLightSensor**( ): TYLightSensor |
| vb | function **nextLightSensor**( ) As YLightSensor |
| cs | YLightSensor **nextLightSensor**( ) |
| java | YLightSensor **nextLightSensor**( ) |
| py | def **nextLightSensor**( ) |

**Returns :**

a pointer to a `YLightSensor` object, corresponding to a light sensor currently online, or a `null` pointer if there are no more light sensors to enumerate.

## lightsensor→registerTimedReportCallback() [lightsensor registerTimedReportCallback: ] YLightSensor

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YLightSensorTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YLightSensorTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYLightSensorTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## lightsensor→registerValueCallback()[lightsensor registerValueCallback: ] YLightSensor

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YLightSensorValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YLightSensorValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYLightSensorValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## lightsensor→set_highestValue() lightsensor→setHighestValue()[lightsensor setHighestValue: ] YLightSensor

Changes the recorded maximal value observed for the ambiant light.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YLightSensor **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the ambiant light

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→set_logFrequency() lightsensor→setLogFrequency()[lightsensor setLogFrequency: ] YLightSensor

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YLightSensor **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→set_logicalName() lightsensor→setLogicalName()[lightsensor setLogicalName: ] YLightSensor

Changes the logical name of the light sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YLightSensor **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the light sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## lightsensor→set_lowestValue() lightsensor→setLowestValue()[lightsensor setLowestValue: ] YLightSensor

Changes the recorded minimal value observed for the ambiant light.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YLightSensor **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the ambiant light

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→set_reportFrequency() lightsensor→setReportFrequency()[lightsensor setReportFrequency: ] YLightSensor

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YLightSensor **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→set_resolution() lightsensor→setResolution()[lightsensor setResolution: ] YLightSensor

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YLightSensor **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## lightsensor→set_userData() lightsensor→setUserData()[lightsensor setUserData: ] YLightSensor

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## lightsensor→wait_async() YLightSensor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.23. Magnetometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_magnetometer.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YMagnetometer = yoctolib.YMagnetometer; |
| php | require_once('yocto_magnetometer.php'); |
| cpp | #include "yocto_magnetometer.h" |
| m | #import "yocto_magnetometer.h" |
| pas | uses yocto_magnetometer; |
| vb | yocto_magnetometer.vb |
| cs | yocto_magnetometer.cs |
| java | import com.yoctopuce.YoctoAPI.YMagnetometer; |
| py | from yocto_magnetometer import * |

**Global functions**

**yFindMagnetometer(func)**
Retrieves a magnetometer for a given identifier.

**yFirstMagnetometer()**
Starts the enumeration of magnetometers currently accessible.

**`YMagnetometer` methods**

**magnetometer→calibrateFromPoints(rawValues, refValues)**
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**magnetometer→describe()**
Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**magnetometer→get_advertisedValue()**
Returns the current value of the magnetometer (no more than 6 characters).

**magnetometer→get_currentRawValue()**
Returns the uncalibrated, unrounded raw value returned by the sensor.

**magnetometer→get_currentValue()**
Returns the current value of the magnetic field.

**magnetometer→get_errorMessage()**
Returns the error message of the latest error with the magnetometer.

**magnetometer→get_errorType()**
Returns the numerical error code of the latest error with the magnetometer.

**magnetometer→get_friendlyName()**
Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.

**magnetometer→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**magnetometer→get_functionId()**
Returns the hardware identifier of the magnetometer, without reference to the module.

**magnetometer→get_hardwareId()**
Returns the unique hardware identifier of the magnetometer in the form `SERIAL.FUNCTIONID`.

**magnetometer→get_highestValue()**

Returns the maximal value observed for the magnetic field since the device was started.

**magnetometer→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**magnetometer→get_logicalName()**

Returns the logical name of the magnetometer.

**magnetometer→get_lowestValue()**

Returns the minimal value observed for the magnetic field since the device was started.

**magnetometer→get_module()**

Gets the `YModule` object for the device on which the function is located.

**magnetometer→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**magnetometer→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**magnetometer→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**magnetometer→get_resolution()**

Returns the resolution of the measured values.

**magnetometer→get_unit()**

Returns the measuring unit for the magnetic field.

**magnetometer→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**magnetometer→get_xValue()**

Returns the X component of the magnetic field, as a floating point number.

**magnetometer→get_yValue()**

Returns the Y component of the magnetic field, as a floating point number.

**magnetometer→get_zValue()**

Returns the Z component of the magnetic field, as a floating point number.

**magnetometer→isOnline()**

Checks if the magnetometer is currently reachable, without raising any error.

**magnetometer→isOnline_async(callback, context)**

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

**magnetometer→load(msValidity)**

Preloads the magnetometer cache with a specified validity duration.

**magnetometer→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**magnetometer→load_async(msValidity, callback, context)**

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

**magnetometer→nextMagnetometer()**

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

**magnetometer→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**magnetometer→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**magnetometer→set_highestValue(newval)**

Changes the recorded maximal value observed.

**magnetometer→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**magnetometer→set_logicalName(newval)**

Changes the logical name of the magnetometer.

**magnetometer→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**magnetometer→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**magnetometer→set_resolution(newval)**

Changes the resolution of the measured physical values.

**magnetometer→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**magnetometer→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YMagnetometer.FindMagnetometer() yFindMagnetometer()yFindMagnetometer()

**YMagnetometer**

Retrieves a magnetometer for a given identifier.

| | |
|---|---|
| js | function **yFindMagnetometer**( **func**) |
| nodejs | function **FindMagnetometer**( **func**) |
| php | function **yFindMagnetometer**( **$func**) |
| cpp | YMagnetometer* **yFindMagnetometer**( const string& **func**) |
| m | YMagnetometer* **yFindMagnetometer**( NSString* **func**) |
| pas | function **yFindMagnetometer**( **func**: string): TYMagnetometer |
| vb | function **yFindMagnetometer**( ByVal **func** As String) As YMagnetometer |
| cs | YMagnetometer **FindMagnetometer**( string **func**) |
| java | YMagnetometer **FindMagnetometer**( String **func**) |
| py | def **FindMagnetometer**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMagnetometer.isOnline()` to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the magnetometer

**Returns :**

a `YMagnetometer` object allowing you to drive the magnetometer.

## YMagnetometer.FirstMagnetometer() yFirstMagnetometer()yFirstMagnetometer() YMagnetometer

Starts the enumeration of magnetometers currently accessible.

| | |
|---|---|
| js | function **yFirstMagnetometer**( ) |
| nodejs | function **FirstMagnetometer**( ) |
| php | function **yFirstMagnetometer**( ) |
| cpp | YMagnetometer* **yFirstMagnetometer**( ) |
| m | YMagnetometer* **yFirstMagnetometer**( ) |
| pas | function **yFirstMagnetometer**( ): TYMagnetometer |
| vb | function **yFirstMagnetometer**( ) As YMagnetometer |
| cs | YMagnetometer **FirstMagnetometer**( ) |
| java | YMagnetometer **FirstMagnetometer**( ) |
| py | def **FirstMagnetometer**( ) |

Use the method `YMagnetometer.nextMagnetometer()` to iterate on next magnetometers.

**Returns :**
a pointer to a `YMagnetometer` object, corresponding to the first magnetometer currently online, or a `null` pointer if there are none.

## magnetometer→calibrateFromPoints() [magnetometer calibrateFromPoints: ] — YMagnetometer

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**,<br>vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues**<br>: (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray,<br>**refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**,<br>List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**,<br>ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YMagnetometer **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→describe()[magnetometer describe] YMagnetometer

Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the magnetometer (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## magnetometer→get_advertisedValue() magnetometer→advertisedValue()[magnetometer advertisedValue] YMagnetometer

Returns the current value of the magnetometer (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YMagnetometer **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the magnetometer (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## magnetometer→get_currentRawValue() magnetometer→currentRawValue()[magnetometer currentRawValue] YMagnetometer

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YMagnetometer **target** **get_currentRawValue** |

**Returns :**
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## magnetometer→get_currentValue() magnetometer→currentValue()[magnetometer currentValue] YMagnetometer

Returns the current value of the magnetic field.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YMagnetometer **target** **get_currentValue** |

**Returns :**
a floating point number corresponding to the current value of the magnetic field

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## magnetometer→get_errorMessage() magnetometer→errorMessage()[magnetometer errorMessage] YMagnetometer

Returns the error message of the latest error with the magnetometer.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the magnetometer object

## magnetometer→get_errorType() magnetometer→errorType() YMagnetometer

Returns the numerical error code of the latest error with the magnetometer.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the magnetometer object

## magnetometer→get_friendlyName()<br>magnetometer→friendlyName()[magnetometer friendlyName] YMagnetometer

Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| `js` | function **get_friendlyName**( ) |
| `nodejs` | function **get_friendlyName**( ) |
| `php` | function **get_friendlyName**( ) |
| `cpp` | string **get_friendlyName**( ) |
| `m` | -(NSString*) friendlyName |
| `cs` | string **get_friendlyName**( ) |
| `java` | String **get_friendlyName**( ) |
| `py` | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the magnetometer if they are defined, otherwise the serial number of the module and the hardware identifier of the magnetometer (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the magnetometer using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## magnetometer→get_functionDescriptor() magnetometer→functionDescriptor()[magnetometer functionDescriptor] YMagnetometer

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## magnetometer→get_functionId() magnetometer→functionId()[magnetometer functionId] YMagnetometer

Returns the hardware identifier of the magnetometer, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the magnetometer (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## magnetometer→get_hardwareId() magnetometer→hardwareId()[magnetometer hardwareId]  YMagnetometer

Returns the unique hardware identifier of the magnetometer in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the magnetometer. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the magnetometer (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## magnetometer→get_highestValue() magnetometer→highestValue()[magnetometer highestValue] YMagnetometer

Returns the maximal value observed for the magnetic field since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YMagnetometer **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## magnetometer→get_logFrequency() magnetometer→logFrequency()[magnetometer logFrequency] YMagnetometer

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YMagnetometer **target** **get_logFrequency** |

**Returns :**
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## magnetometer→get_logicalName() magnetometer→logicalName()[magnetometer logicalName] YMagnetometer

Returns the logical name of the magnetometer.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YMagnetometer **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the magnetometer. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## magnetometer→get_lowestValue() magnetometer→lowestValue()[magnetometer lowestValue] YMagnetometer

Returns the minimal value observed for the magnetic field since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YMagnetometer **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## magnetometer→get_module() magnetometer→module()[magnetometer module] YMagnetometer

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## magnetometer→get_module_async() magnetometer→module_async() YMagnetometer

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## magnetometer→get_recordedData() magnetometer→recordedData()[magnetometer recordedData: ] YMagnetometer

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YMagnetometer **target get_recordedData startTime endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## magnetometer→get_reportFrequency() magnetometer→reportFrequency()[magnetometer reportFrequency] YMagnetometer

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YMagnetometer **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## magnetometer→get_resolution() magnetometer→resolution()[magnetometer resolution] YMagnetometer

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YMagnetometer **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## magnetometer→get_unit() magnetometer→unit()[magnetometer unit] YMagnetometer

Returns the measuring unit for the magnetic field.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YMagnetometer **target** **get_unit** |

**Returns :**

a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## magnetometer→get_userData() magnetometer→userData()[magnetometer userData] YMagnetometer

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## magnetometer→get_xValue() magnetometer→xValue()[magnetometer xValue] YMagnetometer

Returns the X component of the magnetic field, as a floating point number.

| | |
|---|---|
| js | function **get_xValue**( ) |
| nodejs | function **get_xValue**( ) |
| php | function **get_xValue**( ) |
| cpp | double **get_xValue**( ) |
| m | -(double) xValue |
| pas | function **get_xValue**( ): double |
| vb | function **get_xValue**( ) As Double |
| cs | double **get_xValue**( ) |
| java | double **get_xValue**( ) |
| py | def **get_xValue**( ) |
| cmd | YMagnetometer **target** **get_xValue** |

**Returns :**
a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns `Y_XVALUE_INVALID`.

## magnetometer→get_yValue() magnetometer→yValue()[magnetometer yValue] YMagnetometer

Returns the Y component of the magnetic field, as a floating point number.

| | |
|---|---|
| js | function **get_yValue**( ) |
| nodejs | function **get_yValue**( ) |
| php | function **get_yValue**( ) |
| cpp | double **get_yValue**( ) |
| m | -(double) yValue |
| pas | function **get_yValue**( ): double |
| vb | function **get_yValue**( ) As Double |
| cs | double **get_yValue**( ) |
| java | double **get_yValue**( ) |
| py | def **get_yValue**( ) |
| cmd | YMagnetometer **target** **get_yValue** |

**Returns :**
a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

## magnetometer→get_zValue() magnetometer→zValue()[magnetometer zValue] YMagnetometer

Returns the Z component of the magnetic field, as a floating point number.

| | |
|---|---|
| js | function **get_zValue**( ) |
| nodejs | function **get_zValue**( ) |
| php | function **get_zValue**( ) |
| cpp | double **get_zValue**( ) |
| m | -(double) zValue |
| pas | function **get_zValue**( ): double |
| vb | function **get_zValue**( ) As Double |
| cs | double **get_zValue**( ) |
| java | double **get_zValue**( ) |
| py | def **get_zValue**( ) |
| cmd | YMagnetometer **target** **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.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the magnetometer.

**Returns :**

`true` if the magnetometer can be reached, and `false` otherwise

## magnetometer→isOnline_async() YMagnetometer

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## magnetometer→load()[magnetometer load: ] YMagnetometer

Preloads the magnetometer cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## magnetometer→loadCalibrationPoints() [magnetometer loadCalibrationPoints: ] YMagnetometer

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**,<br>vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues**<br>: (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray,<br>var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**,<br>List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**,<br>ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YMagnetometer **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→load_async() YMagnetometer

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## magnetometer→nextMagnetometer()[magnetometer nextMagnetometer] YMagnetometer

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

| | |
|---|---|
| js | function **nextMagnetometer**( ) |
| nodejs | function **nextMagnetometer**( ) |
| php | function **nextMagnetometer**( ) |
| cpp | YMagnetometer * **nextMagnetometer**( ) |
| m | -(YMagnetometer*) **nextMagnetometer** |
| pas | function **nextMagnetometer**( ): TYMagnetometer |
| vb | function **nextMagnetometer**( ) As YMagnetometer |
| cs | YMagnetometer **nextMagnetometer**( ) |
| java | YMagnetometer **nextMagnetometer**( ) |
| py | def **nextMagnetometer**( ) |

**Returns :**

a pointer to a `YMagnetometer` object, corresponding to a magnetometer currently online, or a `null` pointer if there are no more magnetometers to enumerate.

## magnetometer→registerTimedReportCallback() [magnetometer registerTimedReportCallback: ] YMagnetometer

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YMagnetometerTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YMagnetometerTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYMagnetometerTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## magnetometer→registerValueCallback() [magnetometer registerValueCallback: ] YMagnetometer

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YMagnetometerValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YMagnetometerValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYMagnetometerValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## magnetometer→set_highestValue() magnetometer→setHighestValue()[magnetometer setHighestValue: ] YMagnetometer

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YMagnetometer **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→set_logFrequency() magnetometer→setLogFrequency()[magnetometer setLogFrequency: ] YMagnetometer

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YMagnetometer **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→set_logicalName() magnetometer→setLogicalName()[magnetometer setLogicalName: ] YMagnetometer

Changes the logical name of the magnetometer.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YMagnetometer **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the magnetometer.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## magnetometer→set_lowestValue() magnetometer→setLowestValue()[magnetometer setLowestValue: ] YMagnetometer

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YMagnetometer **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→set_reportFrequency() magnetometer→setReportFrequency() [magnetometer setReportFrequency: ] — YMagnetometer

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YMagnetometer **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→set_resolution() magnetometer→setResolution()[magnetometer setResolution: ] YMagnetometer

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YMagnetometer **target set_resolution newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## magnetometer→set_userData() magnetometer→setUserData()[magnetometer setUserData: ] YMagnetometer

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## magnetometer→wait_async() YMagnetometer

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.24. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>var YModule = yoctolib.YModule; |
| php | require_once('yocto_api.php'); |
| cpp | #include "yocto_api.h" |
| m | #import "yocto_api.h" |
| pas | uses yocto_api; |
| vb | yocto_api.vb |
| cs | yocto_api.cs |
| java | import com.yoctopuce.YoctoAPI.YModule; |
| py | from yocto_api import * |

| YMeasure methods |
|---|
| **measure→get_averageValue()**<br>Returns the average value observed during the time interval covered by this measure. |
| **measure→get_endTimeUTC()**<br>Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp). |
| **measure→get_maxValue()**<br>Returns the largest value observed during the time interval covered by this measure. |
| **measure→get_minValue()**<br>Returns the smallest value observed during the time interval covered by this measure. |
| **measure→get_startTimeUTC()**<br>Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp). |

## measure→get_averageValue() measure→averageValue()[measure averageValue] YMeasure

Returns the average value observed during the time interval covered by this measure.

| | |
|---|---|
| js | function **get_averageValue**( ) |
| nodejs | function **get_averageValue**( ) |
| php | function **get_averageValue**( ) |
| cpp | double **get_averageValue**( ) |
| m | -(double) averageValue |
| pas | function **get_averageValue**( ): double |
| vb | function **get_averageValue**( ) As Double |
| cs | double **get_averageValue**( ) |
| java | double **get_averageValue**( ) |
| py | def **get_averageValue**( ) |

**Returns :**
a floating-point number corresponding to the average value observed.

## measure→get_endTimeUTC() measure→endTimeUTC()[measure endTimeUTC] YMeasure

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

| | |
|---|---|
| js | function **get_endTimeUTC**( ) |
| nodejs | function **get_endTimeUTC**( ) |
| php | function **get_endTimeUTC**( ) |
| cpp | double **get_endTimeUTC**( ) |
| m | -(double) endTimeUTC |
| pas | function **get_endTimeUTC**( ): double |
| vb | function **get_endTimeUTC**( ) As Double |
| cs | double **get_endTimeUTC**( ) |
| java | double **get_endTimeUTC**( ) |
| py | def **get_endTimeUTC**( ) |

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

## measure→get_maxValue() measure→maxValue()[measure maxValue] YMeasure

Returns the largest value observed during the time interval covered by this measure.

| | |
|---|---|
| js | function **get_maxValue**( ) |
| nodejs | function **get_maxValue**( ) |
| php | function **get_maxValue**( ) |
| cpp | double **get_maxValue**( ) |
| m | -(double) maxValue |
| pas | function **get_maxValue**( ): double |
| vb | function **get_maxValue**( ) As Double |
| cs | double **get_maxValue**( ) |
| java | double **get_maxValue**( ) |
| py | def **get_maxValue**( ) |

**Returns :**
a floating-point number corresponding to the largest value observed.

## measure→get_minValue() measure→minValue()[measure minValue] YMeasure

Returns the smallest value observed during the time interval covered by this measure.

| | |
|---|---|
| js | function **get_minValue**( ) |
| nodejs | function **get_minValue**( ) |
| php | function **get_minValue**( ) |
| cpp | double **get_minValue**( ) |
| m | -(double) minValue |
| pas | function **get_minValue**( ): double |
| vb | function **get_minValue**( ) As Double |
| cs | double **get_minValue**( ) |
| java | double **get_minValue**( ) |
| py | def **get_minValue**( ) |

**Returns :**

a floating-point number corresponding to the smallest value observed.

## measure→get_startTimeUTC() measure→startTimeUTC()[measure startTimeUTC] YMeasure

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

| | |
|---|---|
| js | function **get_startTimeUTC**( ) |
| nodejs | function **get_startTimeUTC**( ) |
| php | function **get_startTimeUTC**( ) |
| cpp | double **get_startTimeUTC**( ) |
| m | -(double) startTimeUTC |
| pas | function **get_startTimeUTC**( ): double |
| vb | function **get_startTimeUTC**( ) As Double |
| cs | double **get_startTimeUTC**( ) |
| java | double **get_startTimeUTC**( ) |
| py | def **get_startTimeUTC**( ) |

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## 3.25. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>var YModule = yoctolib.YModule; |
| php | require_once('yocto_api.php'); |
| cpp | #include "yocto_api.h" |
| m | #import "yocto_api.h" |
| pas | uses yocto_api; |
| vb | yocto_api.vb |
| cs | yocto_api.cs |
| java | import com.yoctopuce.YoctoAPI.YModule; |
| py | from yocto_api import * |

**Global functions**

**yFindModule(func)**

Allows you to find a module from its serial number or from its logical name.

**yFirstModule()**

Starts the enumeration of modules currently accessible.

**`YModule` methods**

**module→describe()**

Returns a descriptive text that identifies the module.

**module→download(pathname)**

Downloads the specified built-in file and returns a binary buffer with its content.

**module→functionCount()**

Returns the number of functions (beside the "module" interface) available on the module.

**module→functionId(functionIndex)**

Retrieves the hardware identifier of the *n*th function on the module.

**module→functionName(functionIndex)**

Retrieves the logical name of the *n*th function on the module.

**module→functionValue(functionIndex)**

Retrieves the advertised value of the *n*th function on the module.

**module→get_beacon()**

Returns the state of the localization beacon.

**module→get_errorMessage()**

Returns the error message of the latest error with this module object.

**module→get_errorType()**

Returns the numerical error code of the latest error with this module object.

**module→get_firmwareRelease()**

Returns the version of the firmware embedded in the module.

**module→get_hardwareId()**

Returns the unique hardware identifier of the module.

**module→get_icon2d()**

Returns the icon of the module.

**module→get_lastLogs()**

Returns a string with last logs of the module.

**module→get_logicalName()**

Returns the logical name of the module.

**module→get_luminosity()**

Returns the luminosity of the module informative leds (from 0 to 100).

**module→get_persistentSettings()**

Returns the current state of persistent module settings.

**module→get_productId()**

Returns the USB device identifier of the module.

**module→get_productName()**

Returns the commercial name of the module, as set by the factory.

**module→get_productRelease()**

Returns the hardware release version of the module.

**module→get_rebootCountdown()**

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

**module→get_serialNumber()**

Returns the serial number of the module, as set by the factory.

**module→get_upTime()**

Returns the number of milliseconds spent since the module was powered on.

**module→get_usbBandwidth()**

Returns the number of USB interfaces used by the module.

**module→get_usbCurrent()**

Returns the current consumed by the module on the USB bus, in milli-amps.

**module→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**module→isOnline()**

Checks if the module is currently reachable, without raising any error.

**module→isOnline_async(callback, context)**

Checks if the module is currently reachable, without raising any error.

**module→load(msValidity)**

Preloads the module cache with a specified validity duration.

**module→load_async(msValidity, callback, context)**

Preloads the module cache with a specified validity duration (asynchronous version).

**module→nextModule()**

Continues the module enumeration started using `yFirstModule()`.

**module→reboot(secBeforeReboot)**

Schedules a simple module reboot after the given number of seconds.

**module→registerLogCallback(callback)**

todo

**module→revertFromFlash()**

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

**module→saveToFlash()**

Saves current settings in the nonvolatile memory of the module.

**module→set_beacon(newval)**
Turns on or off the module localization beacon.

**module→set_logicalName(newval)**
Changes the logical name of the module.

**module→set_luminosity(newval)**
Changes the luminosity of the module informative leds.

**module→set_usbBandwidth(newval)**
Changes the number of USB interfaces used by the module.

**module→set_userData(data)**
Stores a user context provided as argument in the userData attribute of the function.

**module→triggerFirmwareUpdate(secBeforeReboot)**
Schedules a module reboot into special firmware update mode.

**module→wait_async(callback, context)**
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YModule.FindModule() yFindModule()yFindModule() YModule

Allows you to find a module from its serial number or from its logical name.

| | |
|---|---|
| js | function **yFindModule**( **func**) |
| nodejs | function **FindModule**( **func**) |
| php | function **yFindModule**( **$func**) |
| cpp | YModule* **yFindModule**( string **func**) |
| m | +(YModule*) **yFindModule** : (NSString*) **func** |
| pas | function **yFindModule**( **func**: string): TYModule |
| vb | function **yFindModule**( ByVal **func** As String) As YModule |
| cs | YModule **FindModule**( string **func**) |
| java | YModule **FindModule**( String **func**) |
| py | def **FindModule**( **func**) |

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string containing either the serial number or the logical name of the desired module

**Returns :**

a `YModule` object allowing you to drive the module or get additional information on the module.

## YModule.FirstModule() yFirstModule()yFirstModule() YModule

Starts the enumeration of modules currently accessible.

| | |
|---|---|
| js | function **yFirstModule**( ) |
| nodejs | function **FirstModule**( ) |
| php | function **yFirstModule**( ) |
| cpp | YModule* **yFirstModule**( ) |
| m | YModule* **yFirstModule**( ) |
| pas | function **yFirstModule**( ): TYModule |
| vb | function **yFirstModule**( ) As YModule |
| cs | YModule **FirstModule**( ) |
| java | YModule **FirstModule**( ) |
| py | def **FirstModule**( ) |

Use the method `YModule.nextModule()` to iterate on the next modules.

**Returns :**

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

## module→describe()[module describe] YModule

Returns a descriptive text that identifies the module.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

The text may include either the logical name or the serial number of the module.

**Returns :**
a string that describes the module

## module→download()[module download: ] YModule

Downloads the specified built-in file and returns a binary buffer with its content.

| | |
|---|---|
| js | function **download**( **pathname**) |
| nodejs | function **download**( **pathname**) |
| php | function **download**( **$pathname**) |
| cpp | string **download**( string **pathname**) |
| m | -(NSData*) **download** : (NSString*) **pathname** |
| pas | function **download**( **pathname**: string): TByteArray |
| vb | function **download**( ) As Byte |
| py | def **download**( **pathname**) |
| cmd | YModule **target** **download** **pathname** |

**Parameters :**

**pathname** name of the new file to load

**Returns :**

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

## module→functionCount()[module functionCount] YModule

Returns the number of functions (beside the "module" interface) available on the module.

| | |
|---|---|
| js | function **functionCount**( ) |
| nodejs | function **functionCount**( ) |
| php | function **functionCount**( ) |
| cpp | int **functionCount**( ) |
| m | -(int) **functionCount** |
| pas | function **functionCount**( ): integer |
| vb | function **functionCount**( ) As Integer |
| cs | int **functionCount**( ) |
| py | def **functionCount**( ) |

**Returns :**

the number of functions on the module

On failure, throws an exception or returns a negative error code.

## module→functionId()[module functionId: ] YModule

Retrieves the hardware identifier of the *n*th function on the module.

| | |
|---|---|
| js | function **functionId**( **functionIndex**) |
| nodejs | function **functionId**( **functionIndex**) |
| php | function **functionId**( **$functionIndex**) |
| cpp | string **functionId**( int **functionIndex**) |
| m | -(NSString*) **functionId** : (int) **functionIndex** |
| pas | function **functionId**( **functionIndex**: integer): string |
| vb | function **functionId**( ByVal **functionIndex** As Integer) As String |
| cs | string **functionId**( int **functionIndex**) |
| py | def **functionId**( **functionIndex**) |

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

## module→functionName()[module functionName: ] YModule

Retrieves the logical name of the *n*th function on the module.

| | |
|---|---|
| js | function **functionName**( **functionIndex**) |
| nodejs | function **functionName**( **functionIndex**) |
| php | function **functionName**( **$functionIndex**) |
| cpp | string **functionName**( int **functionIndex**) |
| m | -(NSString*) **functionName** : (int) **functionIndex** |
| pas | function **functionName**( **functionIndex**: integer): string |
| vb | function **functionName**( ByVal **functionIndex** As Integer) As String |
| cs | string **functionName**( int **functionIndex**) |
| py | def **functionName**( **functionIndex**) |

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

## module→functionValue()[module functionValue: ] YModule

Retrieves the advertised value of the *n*th function on the module.

| | |
|---|---|
| js | function **functionValue**( **functionIndex**) |
| nodejs | function **functionValue**( **functionIndex**) |
| php | function **functionValue**( **$functionIndex**) |
| cpp | string **functionValue**( int **functionIndex**) |
| m | -(NSString*) **functionValue** : (int) **functionIndex** |
| pas | function **functionValue**( **functionIndex**: integer): string |
| vb | function **functionValue**( ByVal **functionIndex** As Integer) As String |
| cs | string **functionValue**( int **functionIndex**) |
| py | def **functionValue**( **functionIndex**) |

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

## module→get_beacon() module→beacon()[module beacon] YModule

Returns the state of the localization beacon.

| | |
|---|---|
| js | function **get_beacon**( ) |
| nodejs | function **get_beacon**( ) |
| php | function **get_beacon**( ) |
| cpp | Y_BEACON_enum **get_beacon**( ) |
| m | -(Y_BEACON_enum) beacon |
| pas | function **get_beacon**( ): Integer |
| vb | function **get_beacon**( ) As Integer |
| cs | int **get_beacon**( ) |
| java | int **get_beacon**( ) |
| py | def **get_beacon**( ) |
| cmd | YModule **target** **get_beacon** |

**Returns :**

either `Y_BEACON_OFF` or `Y_BEACON_ON`, according to the state of the localization beacon

On failure, throws an exception or returns `Y_BEACON_INVALID`.

## module→get_errorMessage() module→errorMessage()[module errorMessage] YModule

Returns the error message of the latest error with this module object.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using this module object

## module→get_errorType() module→errorType()

**YModule**

Returns the numerical error code of the latest error with this module object.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using this module object

## module→get_firmwareRelease() module→firmwareRelease()[module firmwareRelease] YModule

Returns the version of the firmware embedded in the module.

| | |
|---|---|
| js | function **get_firmwareRelease**( ) |
| nodejs | function **get_firmwareRelease**( ) |
| php | function **get_firmwareRelease**( ) |
| cpp | string **get_firmwareRelease**( ) |
| m | -(NSString*) firmwareRelease |
| pas | function **get_firmwareRelease**( ): string |
| vb | function **get_firmwareRelease**( ) As String |
| cs | string **get_firmwareRelease**( ) |
| java | String **get_firmwareRelease**( ) |
| py | def **get_firmwareRelease**( ) |
| cmd | YModule **target** **get_firmwareRelease** |

**Returns :**
a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

## module→get_hardwareId() module→hardwareId()[module hardwareId] YModule

Returns the unique hardware identifier of the module.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is made of the device serial number followed by string ".module".

**Returns :**

a string that uniquely identifies the module

## module→get_icon2d()<br>module→icon2d()[module icon2d] YModule

Returns the icon of the module.

| | |
|---|---|
| js | function **get_icon2d**( ) |
| nodejs | function **get_icon2d**( ) |
| php | function **get_icon2d**( ) |
| cpp | string **get_icon2d**( ) |
| m | -(NSData*) icon2d |
| pas | function **get_icon2d**( ): TByteArray |
| vb | function **get_icon2d**( ) As Byte |
| py | def **get_icon2d**( ) |
| cmd | YModule **target** **get_icon2d** |

The icon is a PNG image and does not exceeds 1536 bytes.

**Returns :**
a binary buffer with module icon, in png format.

## module→get_lastLogs() module→lastLogs()[module lastLogs] YModule

Returns a string with last logs of the module.

| | |
|---|---|
| js | function **get_lastLogs**( ) |
| nodejs | function **get_lastLogs**( ) |
| php | function **get_lastLogs**( ) |
| cpp | string **get_lastLogs**( ) |
| m | -(NSString*) lastLogs |
| pas | function **get_lastLogs**( ): string |
| vb | function **get_lastLogs**( ) As String |
| cs | string **get_lastLogs**( ) |
| java | String **get_lastLogs**( ) |
| py | def **get_lastLogs**( ) |
| cmd | YModule **target** **get_lastLogs** |

This method return only logs that are still in the module.

**Returns :**
a string with last logs of the module.

## module→get_logicalName() module→logicalName()[module logicalName] YModule

Returns the logical name of the module.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YModule **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the module

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## module→get_luminosity() module→luminosity()[module luminosity] YModule

Returns the luminosity of the module informative leds (from 0 to 100).

| | |
|---|---|
| js | function **get_luminosity**( ) |
| nodejs | function **get_luminosity**( ) |
| php | function **get_luminosity**( ) |
| cpp | int **get_luminosity**( ) |
| m | -(int) luminosity |
| pas | function **get_luminosity**( ): LongInt |
| vb | function **get_luminosity**( ) As Integer |
| cs | int **get_luminosity**( ) |
| java | int **get_luminosity**( ) |
| py | def **get_luminosity**( ) |
| cmd | YModule **target** **get_luminosity** |

**Returns :**

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

## module→get_persistentSettings() module→persistentSettings()[module persistentSettings] YModule

Returns the current state of persistent module settings.

| | |
|---|---|
| js | function **get_persistentSettings**( ) |
| nodejs | function **get_persistentSettings**( ) |
| php | function **get_persistentSettings**( ) |
| cpp | Y_PERSISTENTSETTINGS_enum **get_persistentSettings**( ) |
| m | -(Y_PERSISTENTSETTINGS_enum) persistentSettings |
| pas | function **get_persistentSettings**( ): Integer |
| vb | function **get_persistentSettings**( ) As Integer |
| cs | int **get_persistentSettings**( ) |
| java | int **get_persistentSettings**( ) |
| py | def **get_persistentSettings**( ) |
| cmd | YModule **target** **get_persistentSettings** |

**Returns :**

a value among `Y_PERSISTENTSETTINGS_LOADED`, `Y_PERSISTENTSETTINGS_SAVED` and `Y_PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `Y_PERSISTENTSETTINGS_INVALID`.

## module→get_productId() module→productId()[module productId] YModule

Returns the USB device identifier of the module.

| | |
|---|---|
| js | function **get_productId**( ) |
| nodejs | function **get_productId**( ) |
| php | function **get_productId**( ) |
| cpp | int **get_productId**( ) |
| m | -(int) productId |
| pas | function **get_productId**( ): LongInt |
| vb | function **get_productId**( ) As Integer |
| cs | int **get_productId**( ) |
| java | int **get_productId**( ) |
| py | def **get_productId**( ) |
| cmd | YModule **target** **get_productId** |

**Returns :**
an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns `Y_PRODUCTID_INVALID`.

## module→get_productName() module→productName()[module productName] YModule

Returns the commercial name of the module, as set by the factory.

| | |
|---|---|
| js | function **get_productName**( ) |
| nodejs | function **get_productName**( ) |
| php | function **get_productName**( ) |
| cpp | string **get_productName**( ) |
| m | -(NSString*) productName |
| pas | function **get_productName**( ): string |
| vb | function **get_productName**( ) As String |
| cs | string **get_productName**( ) |
| java | String **get_productName**( ) |
| py | def **get_productName**( ) |
| cmd | YModule **target** **get_productName** |

**Returns :**
a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `Y_PRODUCTNAME_INVALID`.

## module→get_productRelease() module→productRelease()[module productRelease] YModule

Returns the hardware release version of the module.

| | |
|---|---|
| js | function **get_productRelease**( ) |
| nodejs | function **get_productRelease**( ) |
| php | function **get_productRelease**( ) |
| cpp | int **get_productRelease**( ) |
| m | -(int) productRelease |
| pas | function **get_productRelease**( ): LongInt |
| vb | function **get_productRelease**( ) As Integer |
| cs | int **get_productRelease**( ) |
| java | int **get_productRelease**( ) |
| py | def **get_productRelease**( ) |
| cmd | YModule **target** **get_productRelease** |

**Returns :**
an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns `Y_PRODUCTRELEASE_INVALID`.

## module→get_rebootCountdown() module→rebootCountdown()[module rebootCountdown] YModule

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

| | |
|---|---|
| js | function **get_rebootCountdown**( ) |
| nodejs | function **get_rebootCountdown**( ) |
| php | function **get_rebootCountdown**( ) |
| cpp | int **get_rebootCountdown**( ) |
| m | -(int) rebootCountdown |
| pas | function **get_rebootCountdown**( ): LongInt |
| vb | function **get_rebootCountdown**( ) As Integer |
| cs | int **get_rebootCountdown**( ) |
| java | int **get_rebootCountdown**( ) |
| py | def **get_rebootCountdown**( ) |
| cmd | YModule **target** **get_rebootCountdown** |

**Returns :**

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns `Y_REBOOTCOUNTDOWN_INVALID`.

## module→get_serialNumber() module→serialNumber()[module serialNumber] YModule

Returns the serial number of the module, as set by the factory.

| | |
|---|---|
| js | function **get_serialNumber**( ) |
| nodejs | function **get_serialNumber**( ) |
| php | function **get_serialNumber**( ) |
| cpp | string **get_serialNumber**( ) |
| m | -(NSString*) serialNumber |
| pas | function **get_serialNumber**( ): string |
| vb | function **get_serialNumber**( ) As String |
| cs | string **get_serialNumber**( ) |
| java | String **get_serialNumber**( ) |
| py | def **get_serialNumber**( ) |
| cmd | YModule **target** **get_serialNumber** |

**Returns :**
a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns `Y_SERIALNUMBER_INVALID`.

## module→get_upTime() module→upTime()[module upTime]

**YModule**

Returns the number of milliseconds spent since the module was powered on.

| | |
|---|---|
| js | function **get_upTime**( ) |
| nodejs | function **get_upTime**( ) |
| php | function **get_upTime**( ) |
| cpp | s64 **get_upTime**( ) |
| m | -(s64) upTime |
| pas | function **get_upTime**( ): int64 |
| vb | function **get_upTime**( ) As Long |
| cs | long **get_upTime**( ) |
| java | long **get_upTime**( ) |
| py | def **get_upTime**( ) |
| cmd | YModule **target** **get_upTime** |

**Returns :**

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `Y_UPTIME_INVALID`.

## module→get_usbBandwidth() module→usbBandwidth()[module usbBandwidth] YModule

Returns the number of USB interfaces used by the module.

| | |
|---|---|
| js | function **get_usbBandwidth**( ) |
| nodejs | function **get_usbBandwidth**( ) |
| php | function **get_usbBandwidth**( ) |
| cpp | Y_USBBANDWIDTH_enum **get_usbBandwidth**( ) |
| m | -(Y_USBBANDWIDTH_enum) usbBandwidth |
| pas | function **get_usbBandwidth**( ): Integer |
| vb | function **get_usbBandwidth**( ) As Integer |
| cs | int **get_usbBandwidth**( ) |
| java | int **get_usbBandwidth**( ) |
| py | def **get_usbBandwidth**( ) |
| cmd | YModule **target** **get_usbBandwidth** |

**Returns :**

either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

On failure, throws an exception or returns `Y_USBBANDWIDTH_INVALID`.

## module→get_usbCurrent() module→usbCurrent()[module usbCurrent] YModule

Returns the current consumed by the module on the USB bus, in milli-amps.

| | |
|---|---|
| js | function **get_usbCurrent**( ) |
| nodejs | function **get_usbCurrent**( ) |
| php | function **get_usbCurrent**( ) |
| cpp | int **get_usbCurrent**( ) |
| m | -(int) usbCurrent |
| pas | function **get_usbCurrent**( ): LongInt |
| vb | function **get_usbCurrent**( ) As Integer |
| cs | int **get_usbCurrent**( ) |
| java | int **get_usbCurrent**( ) |
| py | def **get_usbCurrent**( ) |
| cmd | YModule **target** **get_usbCurrent** |

**Returns :**
an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

## module→get_userData() module→userData()[module userData] YModule

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## module→isOnline()[module isOnline] YModule

Checks if the module is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

**Returns :**
`true` if the module can be reached, and `false` otherwise

## module→isOnline_async() YModule

Checks if the module is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## module→load()[module load: ] YModule

Preloads the module cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## module→load_async() YModule

Preloads the module cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded module parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## module→nextModule()[module nextModule] YModule

Continues the module enumeration started using `yFirstModule()`.

| | |
|---|---|
| js | function **nextModule**( ) |
| nodejs | function **nextModule**( ) |
| php | function **nextModule**( ) |
| cpp | YModule * **nextModule**( ) |
| m | -(YModule*) **nextModule** |
| pas | function **nextModule**( ): TYModule |
| vb | function **nextModule**( ) As YModule |
| cs | YModule **nextModule**( ) |
| java | YModule **nextModule**( ) |
| py | def **nextModule**( ) |

**Returns :**

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

## module→reboot()[module reboot: ] YModule

Schedules a simple module reboot after the given number of seconds.

| | |
|---|---|
| js | function **reboot**( **secBeforeReboot**) |
| nodejs | function **reboot**( **secBeforeReboot**) |
| php | function **reboot**( **$secBeforeReboot**) |
| cpp | int **reboot**( int **secBeforeReboot**) |
| m | -(int) **reboot** : (int) **secBeforeReboot** |
| pas | function **reboot**( **secBeforeReboot**: LongInt): LongInt |
| vb | function **reboot**( ) As Integer |
| cs | int **reboot**( int **secBeforeReboot**) |
| java | int **reboot**( int **secBeforeReboot**) |
| py | def **reboot**( **secBeforeReboot**) |
| cmd | YModule **target** **reboot** **secBeforeReboot** |

**Parameters :**

**secBeforeReboot** number of seconds before rebooting

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## module→registerLogCallback()[module registerLogCallback: ] YModule

todo

| | |
|---|---|
| cpp | void **registerLogCallback**( YModuleLogCallback **callback**) |
| m | -(void) **registerLogCallback** : (YModuleLogCallback) **callback** |
| vb | function **registerLogCallback**( ByVal **callback** As YModuleLogCallback) As Integer |
| cs | int **registerLogCallback**( LogCallback **callback**) |
| py | def **registerLogCallback**( **callback**) |

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## module→revertFromFlash()[module revertFromFlash] YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

| | |
|---|---|
| js | function **revertFromFlash**( ) |
| nodejs | function **revertFromFlash**( ) |
| php | function **revertFromFlash**( ) |
| cpp | int **revertFromFlash**( ) |
| m | -(int) **revertFromFlash** |
| pas | function **revertFromFlash**( ): LongInt |
| vb | function **revertFromFlash**( ) As Integer |
| cs | int **revertFromFlash**( ) |
| java | int **revertFromFlash**( ) |
| py | def **revertFromFlash**( ) |
| cmd | YModule **target** **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.

| | |
|---|---|
| js | function **saveToFlash**( ) |
| nodejs | function **saveToFlash**( ) |
| php | function **saveToFlash**( ) |
| cpp | int **saveToFlash**( ) |
| m | -(int) **saveToFlash** |
| pas | function **saveToFlash**( ): LongInt |
| vb | function **saveToFlash**( ) As Integer |
| cs | int **saveToFlash**( ) |
| java | int **saveToFlash**( ) |
| py | def **saveToFlash**( ) |
| cmd | YModule **target** **saveToFlash** |

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## module→set_beacon() module→setBeacon()[module setBeacon: ] YModule

Turns on or off the module localization beacon.

| | |
|---|---|
| js | function **set_beacon**( **newval**) |
| nodejs | function **set_beacon**( **newval**) |
| php | function **set_beacon**( **$newval**) |
| cpp | int **set_beacon**( Y_BEACON_enum **newval**) |
| m | -(int) setBeacon : (Y_BEACON_enum) **newval** |
| pas | function **set_beacon**( **newval**: Integer): integer |
| vb | function **set_beacon**( ByVal **newval** As Integer) As Integer |
| cs | int **set_beacon**( int **newval**) |
| java | int **set_beacon**( int **newval**) |
| py | def **set_beacon**( **newval**) |
| cmd | YModule **target** **set_beacon** **newval** |

**Parameters :**

**newval** either `Y_BEACON_OFF` or `Y_BEACON_ON`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## module→set_logicalName() module→setLogicalName()[module setLogicalName: ] YModule

Changes the logical name of the module.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YModule **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the module

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## module→set_luminosity() module→setLuminosity()[module setLuminosity: ] YModule

Changes the luminosity of the module informative leds.

| | |
|---|---|
| js | function **set_luminosity**( **newval**) |
| nodejs | function **set_luminosity**( **newval**) |
| php | function **set_luminosity**( **$newval**) |
| cpp | int **set_luminosity**( int **newval**) |
| m | -(int) setLuminosity : (int) **newval** |
| pas | function **set_luminosity**( **newval**: LongInt): integer |
| vb | function **set_luminosity**( ByVal **newval** As Integer) As Integer |
| cs | int **set_luminosity**( int **newval**) |
| java | int **set_luminosity**( int **newval**) |
| py | def **set_luminosity**( **newval**) |
| cmd | YModule **target** **set_luminosity** **newval** |

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the luminosity of the module informative leds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## module→set_usbBandwidth() module→setUsbBandwidth()[module setUsbBandwidth: ] YModule

Changes the number of USB interfaces used by the module.

| | |
|---|---|
| js | function **set_usbBandwidth**( **newval**) |
| nodejs | function **set_usbBandwidth**( **newval**) |
| php | function **set_usbBandwidth**( **$newval**) |
| cpp | int **set_usbBandwidth**( Y_USBBANDWIDTH_enum **newval**) |
| m | -(int) setUsbBandwidth : (Y_USBBANDWIDTH_enum) **newval** |
| pas | function **set_usbBandwidth**( **newval**: Integer): integer |
| vb | function **set_usbBandwidth**( ByVal **newval** As Integer) As Integer |
| cs | int **set_usbBandwidth**( int **newval**) |
| java | int **set_usbBandwidth**( int **newval**) |
| py | def **set_usbBandwidth**( **newval**) |
| cmd | YModule **target set_usbBandwidth newval** |

You must reboot the module after changing this setting.

**Parameters :**

**newval** either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## module→set_userData() module→setUserData()[module setUserData: ] YModule

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## module→triggerFirmwareUpdate()[module triggerFirmwareUpdate: ] YModule

Schedules a module reboot into special firmware update mode.

| | |
|---|---|
| js | function **triggerFirmwareUpdate**( **secBeforeReboot**) |
| nodejs | function **triggerFirmwareUpdate**( **secBeforeReboot**) |
| php | function **triggerFirmwareUpdate**( **$secBeforeReboot**) |
| cpp | int **triggerFirmwareUpdate**( int **secBeforeReboot**) |
| m | -(int) **triggerFirmwareUpdate** : (int) **secBeforeReboot** |
| pas | function **triggerFirmwareUpdate**( **secBeforeReboot**: LongInt): LongInt |
| vb | function **triggerFirmwareUpdate**( ) As Integer |
| cs | int **triggerFirmwareUpdate**( int **secBeforeReboot**) |
| java | int **triggerFirmwareUpdate**( int **secBeforeReboot**) |
| py | def **triggerFirmwareUpdate**( **secBeforeReboot**) |
| cmd | YModule **target** **triggerFirmwareUpdate** **secBeforeReboot** |

**Parameters :**

**secBeforeReboot** number of seconds before rebooting

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## module→wait_async() YModule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.26. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_network.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YNetwork = yoctolib.YNetwork; |
| php | require_once('yocto_network.php'); |
| cpp | #include "yocto_network.h" |
| m | #import "yocto_network.h" |
| pas | uses yocto_network; |
| vb | yocto_network.vb |
| cs | yocto_network.cs |
| java | import com.yoctopuce.YoctoAPI.YNetwork; |
| py | from yocto_network import * |

**Global functions**

**yFindNetwork**(**func**)

Retrieves a network interface for a given identifier.

**yFirstNetwork**()

Starts the enumeration of network interfaces currently accessible.

**YNetwork methods**

**network→callbackLogin**(**username**, **password**)

Connects to the notification callback and saves the credentials required to log into it.

**network→describe**()

Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**network→get_adminPassword**()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

**network→get_advertisedValue**()

Returns the current value of the network interface (no more than 6 characters).

**network→get_callbackCredentials**()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

**network→get_callbackEncoding**()

Returns the encoding standard to use for representing notification values.

**network→get_callbackMaxDelay**()

Returns the maximum waiting time between two callback notifications, in seconds.

**network→get_callbackMethod**()

Returns the HTTP method used to notify callbacks for significant state changes.

**network→get_callbackMinDelay**()

Returns the minimum waiting time between two callback notifications, in seconds.

**network→get_callbackUrl**()

Returns the callback URL to notify of significant state changes.

**network→get_discoverable**()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

**network→get_errorMessage()**
Returns the error message of the latest error with the network interface.

**network→get_errorType()**
Returns the numerical error code of the latest error with the network interface.

**network→get_friendlyName()**
Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

**network→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**network→get_functionId()**
Returns the hardware identifier of the network interface, without reference to the module.

**network→get_hardwareId()**
Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

**network→get_ipAddress()**
Returns the IP address currently in use by the device.

**network→get_logicalName()**
Returns the logical name of the network interface.

**network→get_macAddress()**
Returns the MAC address of the network interface.

**network→get_module()**
Gets the `YModule` object for the device on which the function is located.

**network→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**network→get_poeCurrent()**
Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

**network→get_primaryDNS()**
Returns the IP address of the primary name server to be used by the module.

**network→get_readiness()**
Returns the current established working mode of the network interface.

**network→get_router()**
Returns the IP address of the router on the device subnet (default gateway).

**network→get_secondaryDNS()**
Returns the IP address of the secondary name server to be used by the module.

**network→get_subnetMask()**
Returns the subnet mask currently used by the device.

**network→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**network→get_userPassword()**
Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

**network→get_wwwWatchdogDelay()**
Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

**network→isOnline()**
Checks if the network interface is currently reachable, without raising any error.

**network→isOnline_async(callback, context)**
Checks if the network interface is currently reachable, without raising any error (asynchronous version).

**network→load(msValidity)**
Preloads the network interface cache with a specified validity duration.

**network→load_async(msValidity, callback, context)**
Preloads the network interface cache with a specified validity duration (asynchronous version).

**network→nextNetwork()**
Continues the enumeration of network interfaces started using `yFirstNetwork()`.

**network→ping(host)**
Pings str_host to test the network connectivity.

**network→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**network→set_adminPassword(newval)**
Changes the password for the "admin" user.

**network→set_callbackCredentials(newval)**
Changes the credentials required to connect to the callback address.

**network→set_callbackEncoding(newval)**
Changes the encoding standard to use for representing notification values.

**network→set_callbackMaxDelay(newval)**
Changes the maximum waiting time between two callback notifications, in seconds.

**network→set_callbackMethod(newval)**
Changes the HTTP method used to notify callbacks for significant state changes.

**network→set_callbackMinDelay(newval)**
Changes the minimum waiting time between two callback notifications, in seconds.

**network→set_callbackUrl(newval)**
Changes the callback URL to notify significant state changes.

**network→set_discoverable(newval)**
Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

**network→set_logicalName(newval)**
Changes the logical name of the network interface.

**network→set_primaryDNS(newval)**
Changes the IP address of the primary name server to be used by the module.

**network→set_secondaryDNS(newval)**
Changes the IP address of the secondary name server to be used by the module.

**network→set_userData(data)**
Stores a user context provided as argument in the userData attribute of the function.

**network→set_userPassword(newval)**
Changes the password for the "user" user.

**network→set_wwwWatchdogDelay(newval)**
Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

**network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)**
Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

**network→useStaticIP(ipAddress, subnetMaskLen, router)**
Changes the configuration of the network interface to use a static IP address.

**network→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YNetwork.FindNetwork() yFindNetwork()yFindNetwork() YNetwork

Retrieves a network interface for a given identifier.

| | |
|---|---|
| js | function **yFindNetwork**( **func**) |
| nodejs | function **FindNetwork**( **func**) |
| php | function **yFindNetwork**( **$func**) |
| cpp | YNetwork* **yFindNetwork**( const string& **func**) |
| m | YNetwork* **yFindNetwork**( NSString* **func**) |
| pas | function **yFindNetwork**( **func**: string): TYNetwork |
| vb | function **yFindNetwork**( ByVal **func** As String) As YNetwork |
| cs | YNetwork **FindNetwork**( string **func**) |
| java | YNetwork **FindNetwork**( String **func**) |
| py | def **FindNetwork**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the network interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YNetwork.isOnline()` to test if the network interface is indeed online at a given time. In case of ambiguity when looking for a network interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the network interface

**Returns :**

a `YNetwork` object allowing you to drive the network interface.

## YNetwork.FirstNetwork() yFirstNetwork()yFirstNetwork() YNetwork

Starts the enumeration of network interfaces currently accessible.

| | |
|---|---|
| js | function **yFirstNetwork**( ) |
| nodejs | function **FirstNetwork**( ) |
| php | function **yFirstNetwork**( ) |
| cpp | YNetwork* **yFirstNetwork**( ) |
| m | YNetwork* **yFirstNetwork**( ) |
| pas | function **yFirstNetwork**( ): TYNetwork |
| vb | function **yFirstNetwork**( ) As YNetwork |
| cs | YNetwork **FirstNetwork**( ) |
| java | YNetwork **FirstNetwork**( ) |
| py | def **FirstNetwork**( ) |

Use the method `YNetwork.nextNetwork()` to iterate on next network interfaces.

**Returns :**

a pointer to a `YNetwork` object, corresponding to the first network interface currently online, or a `null` pointer if there are none.

## network→callbackLogin()[network callbackLogin: ] YNetwork

Connects to the notification callback and saves the credentials required to log into it.

| | |
|---|---|
| js | function **callbackLogin**( **username**, **password**) |
| nodejs | function **callbackLogin**( **username**, **password**) |
| php | function **callbackLogin**( **$username**, **$password**) |
| cpp | int **callbackLogin**( string **username**, string **password**) |
| m | -(int) **callbackLogin** : (NSString*) **username** : (NSString*) **password** |
| pas | function **callbackLogin**( **username**: string, **password**: string): integer |
| vb | function **callbackLogin**( ByVal **username** As String, ByVal **password** As String) As Integer |
| cs | int **callbackLogin**( string **username**, string **password**) |
| java | int **callbackLogin**( String **username**, String **password**) |
| py | def **callbackLogin**( **username**, **password**) |
| cmd | YNetwork **target** **callbackLogin** **username** **password** |

The password is not stored into the module, only a hashed copy of the credentials are saved. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**username** username required to log to the callback

**password** password required to log to the callback

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→describe()[network describe] YNetwork

Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the network interface (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## network→get_adminPassword() network→adminPassword()[network adminPassword] YNetwork

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

| | |
|---|---|
| js | function **get_adminPassword**( ) |
| nodejs | function **get_adminPassword**( ) |
| php | function **get_adminPassword**( ) |
| cpp | string **get_adminPassword**( ) |
| m | -(NSString*) adminPassword |
| pas | function **get_adminPassword**( ): string |
| vb | function **get_adminPassword**( ) As String |
| cs | string **get_adminPassword**( ) |
| java | String **get_adminPassword**( ) |
| py | def **get_adminPassword**( ) |
| cmd | YNetwork **target** **get_adminPassword** |

**Returns :**
a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns `Y_ADMINPASSWORD_INVALID`.

## network→get_advertisedValue() network→advertisedValue()[network advertisedValue] YNetwork

Returns the current value of the network interface (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YNetwork **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the network interface (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## network→get_callbackCredentials() network→callbackCredentials()[network callbackCredentials] YNetwork

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

| | |
|---|---|
| js | function **get_callbackCredentials**( ) |
| nodejs | function **get_callbackCredentials**( ) |
| php | function **get_callbackCredentials**( ) |
| cpp | string **get_callbackCredentials**( ) |
| m | -(NSString*) callbackCredentials |
| pas | function **get_callbackCredentials**( ): string |
| vb | function **get_callbackCredentials**( ) As String |
| cs | string **get_callbackCredentials**( ) |
| java | String **get_callbackCredentials**( ) |
| py | def **get_callbackCredentials**( ) |
| cmd | YNetwork **target** **get_callbackCredentials** |

**Returns :**

a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns `Y_CALLBACKCREDENTIALS_INVALID`.

## network→get_callbackEncoding() network→callbackEncoding()[network callbackEncoding] YNetwork

Returns the encoding standard to use for representing notification values.

| | |
|---|---|
| js | function **get_callbackEncoding**( ) |
| nodejs | function **get_callbackEncoding**( ) |
| php | function **get_callbackEncoding**( ) |
| cpp | Y_CALLBACKENCODING_enum **get_callbackEncoding**( ) |
| m | -(Y_CALLBACKENCODING_enum) callbackEncoding |
| pas | function **get_callbackEncoding**( ): Integer |
| vb | function **get_callbackEncoding**( ) As Integer |
| cs | int **get_callbackEncoding**( ) |
| java | int **get_callbackEncoding**( ) |
| py | def **get_callbackEncoding**( ) |
| cmd | YNetwork **target** **get_callbackEncoding** |

**Returns :**

a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV` and `Y_CALLBACKENCODING_YOCTO_API` corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns `Y_CALLBACKENCODING_INVALID`.

## network→get_callbackMaxDelay() network→callbackMaxDelay()[network callbackMaxDelay] YNetwork

Returns the maximum waiting time between two callback notifications, in seconds.

| | |
|---|---|
| js | function **get_callbackMaxDelay**( ) |
| nodejs | function **get_callbackMaxDelay**( ) |
| php | function **get_callbackMaxDelay**( ) |
| cpp | int **get_callbackMaxDelay**( ) |
| m | -(int) callbackMaxDelay |
| pas | function **get_callbackMaxDelay**( ): LongInt |
| vb | function **get_callbackMaxDelay**( ) As Integer |
| cs | int **get_callbackMaxDelay**( ) |
| java | int **get_callbackMaxDelay**( ) |
| py | def **get_callbackMaxDelay**( ) |
| cmd | YNetwork **target** **get_callbackMaxDelay** |

**Returns :**

an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns `Y_CALLBACKMAXDELAY_INVALID`.

## network→get_callbackMethod() network→callbackMethod()[network callbackMethod] YNetwork

Returns the HTTP method used to notify callbacks for significant state changes.

| | |
|---|---|
| js | function **get_callbackMethod**( ) |
| nodejs | function **get_callbackMethod**( ) |
| php | function **get_callbackMethod**( ) |
| cpp | Y_CALLBACKMETHOD_enum **get_callbackMethod**( ) |
| m | -(Y_CALLBACKMETHOD_enum) callbackMethod |
| pas | function **get_callbackMethod**( ): Integer |
| vb | function **get_callbackMethod**( ) As Integer |
| cs | int **get_callbackMethod**( ) |
| java | int **get_callbackMethod**( ) |
| py | def **get_callbackMethod**( ) |
| cmd | YNetwork **target** **get_callbackMethod** |

**Returns :**

a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns `Y_CALLBACKMETHOD_INVALID`.

## network→get_callbackMinDelay() network→callbackMinDelay()[network callbackMinDelay] YNetwork

Returns the minimum waiting time between two callback notifications, in seconds.

| | |
|---|---|
| js | function **get_callbackMinDelay**( ) |
| nodejs | function **get_callbackMinDelay**( ) |
| php | function **get_callbackMinDelay**( ) |
| cpp | int **get_callbackMinDelay**( ) |
| m | -(int) callbackMinDelay |
| pas | function **get_callbackMinDelay**( ): LongInt |
| vb | function **get_callbackMinDelay**( ) As Integer |
| cs | int **get_callbackMinDelay**( ) |
| java | int **get_callbackMinDelay**( ) |
| py | def **get_callbackMinDelay**( ) |
| cmd | YNetwork **target** **get_callbackMinDelay** |

**Returns :**
an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns `Y_CALLBACKMINDELAY_INVALID`.

## network→get_callbackUrl() network→callbackUrl()[network callbackUrl] YNetwork

Returns the callback URL to notify of significant state changes.

| | |
|---|---|
| js | function **get_callbackUrl**( ) |
| nodejs | function **get_callbackUrl**( ) |
| php | function **get_callbackUrl**( ) |
| cpp | string **get_callbackUrl**( ) |
| m | -(NSString*) callbackUrl |
| pas | function **get_callbackUrl**( ): string |
| vb | function **get_callbackUrl**( ) As String |
| cs | string **get_callbackUrl**( ) |
| java | String **get_callbackUrl**( ) |
| py | def **get_callbackUrl**( ) |
| cmd | YNetwork **target** **get_callbackUrl** |

**Returns :**
a string corresponding to the callback URL to notify of significant state changes

On failure, throws an exception or returns `Y_CALLBACKURL_INVALID`.

## network→get_discoverable() network→discoverable()[network discoverable]

YNetwork

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

| | |
|---|---|
| js | function **get_discoverable**( ) |
| nodejs | function **get_discoverable**( ) |
| php | function **get_discoverable**( ) |
| cpp | Y_DISCOVERABLE_enum **get_discoverable**( ) |
| m | -(Y_DISCOVERABLE_enum) discoverable |
| pas | function **get_discoverable**( ): Integer |
| vb | function **get_discoverable**( ) As Integer |
| cs | int **get_discoverable**( ) |
| java | int **get_discoverable**( ) |
| py | def **get_discoverable**( ) |
| cmd | YNetwork **target** **get_discoverable** |

**Returns :**

either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns `Y_DISCOVERABLE_INVALID`.

## network→get_errorMessage() network→errorMessage()[network errorMessage] YNetwork

Returns the error message of the latest error with the network interface.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the network interface object

## network→get_errorType() network→errorType()

**YNetwork**

Returns the numerical error code of the latest error with the network interface.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the network interface object

## network→get_friendlyName() YNetwork
## network→friendlyName()[network friendlyName]

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the network interface if they are defined, otherwise the serial number of the module and the hardware identifier of the network interface (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the network interface using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## network→get_functionDescriptor() network→functionDescriptor()[network functionDescriptor] YNetwork

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**
an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## network→get_functionId() network→functionId()[network functionId] YNetwork

Returns the hardware identifier of the network interface, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the network interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## network→get_hardwareId() network→hardwareId()[network hardwareId] YNetwork

Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the network interface. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the network interface (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## network→get_ipAddress() network→ipAddress()[network ipAddress] — YNetwork

Returns the IP address currently in use by the device.

| | |
|---|---|
| js | function **get_ipAddress**( ) |
| nodejs | function **get_ipAddress**( ) |
| php | function **get_ipAddress**( ) |
| cpp | string **get_ipAddress**( ) |
| m | -(NSString*) ipAddress |
| pas | function **get_ipAddress**( ): string |
| vb | function **get_ipAddress**( ) As String |
| cs | string **get_ipAddress**( ) |
| java | String **get_ipAddress**( ) |
| py | def **get_ipAddress**( ) |
| cmd | YNetwork **target** **get_ipAddress** |

The address may have been configured statically, or provided by a DHCP server.

**Returns :**
a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns `Y_IPADDRESS_INVALID`.

## network→get_logicalName() network→logicalName()[network logicalName] YNetwork

Returns the logical name of the network interface.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YNetwork **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the network interface. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## network→get_macAddress() network→macAddress()[network macAddress] YNetwork

Returns the MAC address of the network interface.

| | |
|---|---|
| js | function **get_macAddress**( ) |
| nodejs | function **get_macAddress**( ) |
| php | function **get_macAddress**( ) |
| cpp | string **get_macAddress**( ) |
| m | -(NSString*) macAddress |
| pas | function **get_macAddress**( ): string |
| vb | function **get_macAddress**( ) As String |
| cs | string **get_macAddress**( ) |
| java | String **get_macAddress**( ) |
| py | def **get_macAddress**( ) |
| cmd | YNetwork **target get_macAddress** |

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

**Returns :**
a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns `Y_MACADDRESS_INVALID`.

## network→get_module() network→module()[network module] YNetwork

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## network→get_module_async() network→module_async() YNetwork

Gets the YModule object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## network→get_poeCurrent() network→poeCurrent()[network poeCurrent] YNetwork

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

| | |
|---|---|
| `js` | function **get_poeCurrent**( ) |
| `nodejs` | function **get_poeCurrent**( ) |
| `php` | function **get_poeCurrent**( ) |
| `cpp` | int **get_poeCurrent**( ) |
| `m` | -(int) poeCurrent |
| `pas` | function **get_poeCurrent**( ): LongInt |
| `vb` | function **get_poeCurrent**( ) As Integer |
| `cs` | int **get_poeCurrent**( ) |
| `java` | int **get_poeCurrent**( ) |
| `py` | def **get_poeCurrent**( ) |
| `cmd` | YNetwork **target** **get_poeCurrent** |

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

**Returns :**
an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns `Y_POECURRENT_INVALID`.

## network→get_primaryDNS() network→primaryDNS()[network primaryDNS] YNetwork

Returns the IP address of the primary name server to be used by the module.

| | |
|---|---|
| js | function **get_primaryDNS**( ) |
| nodejs | function **get_primaryDNS**( ) |
| php | function **get_primaryDNS**( ) |
| cpp | string **get_primaryDNS**( ) |
| m | -(NSString*) primaryDNS |
| pas | function **get_primaryDNS**( ): string |
| vb | function **get_primaryDNS**( ) As String |
| cs | string **get_primaryDNS**( ) |
| java | String **get_primaryDNS**( ) |
| py | def **get_primaryDNS**( ) |
| cmd | YNetwork **target** **get_primaryDNS** |

**Returns :**
a string corresponding to the IP address of the primary name server to be used by the module

On failure, throws an exception or returns `Y_PRIMARYDNS_INVALID`.

## network→get_readiness() network→readiness()[network readiness] YNetwork

Returns the current established working mode of the network interface.

| | |
|---|---|
| js | function **get_readiness**( ) |
| nodejs | function **get_readiness**( ) |
| php | function **get_readiness**( ) |
| cpp | Y_READINESS_enum **get_readiness**( ) |
| m | -(Y_READINESS_enum) readiness |
| pas | function **get_readiness**( ): Integer |
| vb | function **get_readiness**( ) As Integer |
| cs | int **get_readiness**( ) |
| java | int **get_readiness**( ) |
| py | def **get_readiness**( ) |
| cmd | YNetwork **target** **get_readiness** |

Level zero (DOWN_0) means that no hardware link has been detected. Either there is no signal on the network cable, or the selected wireless access point cannot be detected. Level 1 (LIVE_1) is reached when the network is detected, but is not yet connected. For a wireless network, this shows that the requested SSID is present. Level 2 (LINK_2) is reached when the hardware connection is established. For a wired network connection, level 2 means that the cable is attached at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS_4) is reached when the DNS server is reachable on the network. Level 5 (WWW_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

**Returns :**

a value among `Y_READINESS_DOWN`, `Y_READINESS_EXISTS`, `Y_READINESS_LINKED`, `Y_READINESS_LAN_OK` and `Y_READINESS_WWW_OK` corresponding to the current established working mode of the network interface

On failure, throws an exception or returns `Y_READINESS_INVALID`.

## network→get_router() network→router()[network router] YNetwork

Returns the IP address of the router on the device subnet (default gateway).

| | |
|---|---|
| js | function **get_router**( ) |
| nodejs | function **get_router**( ) |
| php | function **get_router**( ) |
| cpp | string **get_router**( ) |
| m | -(NSString*) router |
| pas | function **get_router**( ): string |
| vb | function **get_router**( ) As String |
| cs | string **get_router**( ) |
| java | String **get_router**( ) |
| py | def **get_router**( ) |
| cmd | YNetwork **target** **get_router** |

**Returns :**
a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns `Y_ROUTER_INVALID`.

## network→get_secondaryDNS() network→secondaryDNS()[network secondaryDNS] YNetwork

Returns the IP address of the secondary name server to be used by the module.

| | |
|---|---|
| js | function **get_secondaryDNS**( ) |
| nodejs | function **get_secondaryDNS**( ) |
| php | function **get_secondaryDNS**( ) |
| cpp | string **get_secondaryDNS**( ) |
| m | -(NSString*) secondaryDNS |
| pas | function **get_secondaryDNS**( ): string |
| vb | function **get_secondaryDNS**( ) As String |
| cs | string **get_secondaryDNS**( ) |
| java | String **get_secondaryDNS**( ) |
| py | def **get_secondaryDNS**( ) |
| cmd | YNetwork **target** **get_secondaryDNS** |

**Returns :**
a string corresponding to the IP address of the secondary name server to be used by the module

On failure, throws an exception or returns `Y_SECONDARYDNS_INVALID`.

## network→get_subnetMask() network→subnetMask()[network subnetMask] YNetwork

Returns the subnet mask currently used by the device.

| | |
|---|---|
| js | function **get_subnetMask**( ) |
| nodejs | function **get_subnetMask**( ) |
| php | function **get_subnetMask**( ) |
| cpp | string **get_subnetMask**( ) |
| m | -(NSString*) subnetMask |
| pas | function **get_subnetMask**( ): string |
| vb | function **get_subnetMask**( ) As String |
| cs | string **get_subnetMask**( ) |
| java | String **get_subnetMask**( ) |
| py | def **get_subnetMask**( ) |
| cmd | YNetwork **target** **get_subnetMask** |

**Returns :**
a string corresponding to the subnet mask currently used by the device

On failure, throws an exception or returns `Y_SUBNETMASK_INVALID`.

## network→get_userData() network→userData()[network userData] — YNetwork

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| `js` | function **get_userData**( ) |
| `nodejs` | function **get_userData**( ) |
| `php` | function **get_userData**( ) |
| `cpp` | void * **get_userData**( ) |
| `m` | -(void*) userData |
| `pas` | function **get_userData**( ): Tobject |
| `vb` | function **get_userData**( ) As Object |
| `cs` | object **get_userData**( ) |
| `java` | Object **get_userData**( ) |
| `py` | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## network→get_userPassword() network→userPassword()[network userPassword] YNetwork

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

| | |
|---|---|
| js | function **get_userPassword**( ) |
| nodejs | function **get_userPassword**( ) |
| php | function **get_userPassword**( ) |
| cpp | string **get_userPassword**( ) |
| m | -(NSString*) userPassword |
| pas | function **get_userPassword**( ): string |
| vb | function **get_userPassword**( ) As String |
| cs | string **get_userPassword**( ) |
| java | String **get_userPassword**( ) |
| py | def **get_userPassword**( ) |
| cmd | YNetwork **target** **get_userPassword** |

**Returns :**
a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns `Y_USERPASSWORD_INVALID`.

## network→get_wwwWatchdogDelay() network→wwwWatchdogDelay()[network wwwWatchdogDelay] YNetwork

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

| | |
|---|---|
| js | function **get_wwwWatchdogDelay**( ) |
| nodejs | function **get_wwwWatchdogDelay**( ) |
| php | function **get_wwwWatchdogDelay**( ) |
| cpp | int **get_wwwWatchdogDelay**( ) |
| m | -(int) wwwWatchdogDelay |
| pas | function **get_wwwWatchdogDelay**( ): LongInt |
| vb | function **get_wwwWatchdogDelay**( ) As Integer |
| cs | int **get_wwwWatchdogDelay**( ) |
| java | int **get_wwwWatchdogDelay**( ) |
| py | def **get_wwwWatchdogDelay**( ) |
| cmd | YNetwork **target** **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.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

**Returns :**

`true` if the network interface can be reached, and `false` otherwise

## network→isOnline_async() YNetwork

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## network→load()[network load: ] YNetwork

Preloads the network interface cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## network→load_async() YNetwork

Preloads the network interface cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## network→nextNetwork()[network nextNetwork] YNetwork

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

| | |
|---|---|
| js | function **nextNetwork**( ) |
| nodejs | function **nextNetwork**( ) |
| php | function **nextNetwork**( ) |
| cpp | YNetwork * **nextNetwork**( ) |
| m | -(YNetwork*) **nextNetwork** |
| pas | function **nextNetwork**( ): TYNetwork |
| vb | function **nextNetwork**( ) As YNetwork |
| cs | YNetwork **nextNetwork**( ) |
| java | YNetwork **nextNetwork**( ) |
| py | def **nextNetwork**( ) |

**Returns :**

a pointer to a `YNetwork` object, corresponding to a network interface currently online, or a `null` pointer if there are no more network interfaces to enumerate.

## network→ping()[network ping: ] YNetwork

Pings str_host to test the network connectivity.

| | |
|---|---|
| js | function **ping**( **host**) |
| nodejs | function **ping**( **host**) |
| php | function **ping**( **$host**) |
| cpp | string **ping**( string **host**) |
| m | -(NSString*) **ping** : (NSString*) **host** |
| pas | function **ping**( **host**: string): string |
| vb | function **ping**( ) As String |
| cs | string **ping**( string **host**) |
| java | String **ping**( String **host**) |
| py | def **ping**( **host**) |
| cmd | YNetwork **target** **ping** **host** |

Sends four ICMP ECHO_REQUEST requests from the module to the target str_host. This method returns a string with the result of the 4 ICMP ECHO_REQUEST requests.

**Parameters :**

**host** the hostname or the IP address of the target

**Returns :**

a string with the result of the ping.

## network→registerValueCallback()[network registerValueCallback: ] YNetwork

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YNetworkValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YNetworkValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYNetworkValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## network→set_adminPassword() network→setAdminPassword()[network setAdminPassword: ] YNetwork

Changes the password for the "admin" user.

| | |
|---|---|
| js | function **set_adminPassword**( **newval**) |
| nodejs | function **set_adminPassword**( **newval**) |
| php | function **set_adminPassword**( **$newval**) |
| cpp | int **set_adminPassword**( const string& **newval**) |
| m | -(int) setAdminPassword : (NSString*) **newval** |
| pas | function **set_adminPassword**( **newval**: string): integer |
| vb | function **set_adminPassword**( ByVal **newval** As String) As Integer |
| cs | int **set_adminPassword**( string **newval**) |
| java | int **set_adminPassword**( String **newval**) |
| py | def **set_adminPassword**( **newval**) |
| cmd | YNetwork **target set_adminPassword newval** |

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the password for the "admin" user

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_callbackCredentials() network→setCallbackCredentials()[network setCallbackCredentials: ] YNetwork

Changes the credentials required to connect to the callback address.

| | |
|---|---|
| js | function **set_callbackCredentials**( **newval**) |
| nodejs | function **set_callbackCredentials**( **newval**) |
| php | function **set_callbackCredentials**( **$newval**) |
| cpp | int **set_callbackCredentials**( const string& **newval**) |
| m | -(int) setCallbackCredentials : (NSString*) **newval** |
| pas | function **set_callbackCredentials**( **newval**: string): integer |
| vb | function **set_callbackCredentials**( ByVal **newval** As String) As Integer |
| cs | int **set_callbackCredentials**( string **newval**) |
| java | int **set_callbackCredentials**( String **newval**) |
| py | def **set_callbackCredentials**( **newval**) |
| cmd | YNetwork **target** **set_callbackCredentials** **newval** |

The credentials must be provided as returned by function `get_callbackCredentials`, in the form `username:hash`. The method used to compute the hash varies according to the the authentication scheme implemented by the callback, For Basic authentication, the hash is the MD5 of the string `username:password`. For Digest authentication, the hash is the MD5 of the string `username:realm:password`. For a simpler way to configure callback credentials, use function `callbackLogin` instead. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the credentials required to connect to the callback address

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_callbackEncoding() network→setCallbackEncoding()[network setCallbackEncoding: ] YNetwork

Changes the encoding standard to use for representing notification values.

| | |
|---|---|
| js | function **set_callbackEncoding**( **newval**) |
| nodejs | function **set_callbackEncoding**( **newval**) |
| php | function **set_callbackEncoding**( **$newval**) |
| cpp | int **set_callbackEncoding**( Y_CALLBACKENCODING_enum **newval**) |
| m | -(int) setCallbackEncoding : (Y_CALLBACKENCODING_enum) **newval** |
| pas | function **set_callbackEncoding**( **newval**: Integer): integer |
| vb | function **set_callbackEncoding**( ByVal **newval** As Integer) As Integer |
| cs | int **set_callbackEncoding**( int **newval**) |
| java | int **set_callbackEncoding**( int **newval**) |
| py | def **set_callbackEncoding**( **newval**) |
| cmd | YNetwork **target** **set_callbackEncoding** **newval** |

**Parameters :**

**newval** a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV` and `Y_CALLBACKENCODING_YOCTO_API` corresponding to the encoding standard to use for representing notification values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_callbackMaxDelay() network→setCallbackMaxDelay()[network setCallbackMaxDelay: ] YNetwork

Changes the maximum waiting time between two callback notifications, in seconds.

| | |
|---|---|
| js | function **set_callbackMaxDelay**( **newval**) |
| nodejs | function **set_callbackMaxDelay**( **newval**) |
| php | function **set_callbackMaxDelay**( **$newval**) |
| cpp | int **set_callbackMaxDelay**( int **newval**) |
| m | -(int) setCallbackMaxDelay : (int) **newval** |
| pas | function **set_callbackMaxDelay**( **newval**: LongInt): integer |
| vb | function **set_callbackMaxDelay**( ByVal **newval** As Integer) As Integer |
| cs | int **set_callbackMaxDelay**( int **newval**) |
| java | int **set_callbackMaxDelay**( int **newval**) |
| py | def **set_callbackMaxDelay**( **newval**) |
| cmd | YNetwork **target** **set_callbackMaxDelay** **newval** |

**Parameters :**

**newval** an integer corresponding to the maximum waiting time between two callback notifications, in seconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_callbackMethod() network→setCallbackMethod()[network setCallbackMethod: ] YNetwork

Changes the HTTP method used to notify callbacks for significant state changes.

| | |
|---|---|
| js | function **set_callbackMethod**( **newval**) |
| nodejs | function **set_callbackMethod**( **newval**) |
| php | function **set_callbackMethod**( **$newval**) |
| cpp | int **set_callbackMethod**( Y_CALLBACKMETHOD_enum **newval**) |
| m | -(int) setCallbackMethod : (Y_CALLBACKMETHOD_enum) **newval** |
| pas | function **set_callbackMethod**( **newval**: Integer): integer |
| vb | function **set_callbackMethod**( ByVal **newval** As Integer) As Integer |
| cs | int **set_callbackMethod**( int **newval**) |
| java | int **set_callbackMethod**( int **newval**) |
| py | def **set_callbackMethod**( **newval**) |
| cmd | YNetwork **target set_callbackMethod newval** |

**Parameters :**

**newval** a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_callbackMinDelay() network→setCallbackMinDelay()[network setCallbackMinDelay: ] YNetwork

Changes the minimum waiting time between two callback notifications, in seconds.

| | |
|---|---|
| js | function **set_callbackMinDelay**( **newval**) |
| nodejs | function **set_callbackMinDelay**( **newval**) |
| php | function **set_callbackMinDelay**( **$newval**) |
| cpp | int **set_callbackMinDelay**( int **newval**) |
| m | -(int) setCallbackMinDelay : (int) **newval** |
| pas | function **set_callbackMinDelay**( **newval**: LongInt): integer |
| vb | function **set_callbackMinDelay**( ByVal **newval** As Integer) As Integer |
| cs | int **set_callbackMinDelay**( int **newval**) |
| java | int **set_callbackMinDelay**( int **newval**) |
| py | def **set_callbackMinDelay**( **newval**) |
| cmd | YNetwork **target set_callbackMinDelay newval** |

**Parameters :**

**newval** an integer corresponding to the minimum waiting time between two callback notifications, in seconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_callbackUrl() network→setCallbackUrl()[network setCallbackUrl: ] YNetwork

Changes the callback URL to notify significant state changes.

| | |
|---|---|
| js | function **set_callbackUrl**( **newval**) |
| nodejs | function **set_callbackUrl**( **newval**) |
| php | function **set_callbackUrl**( **$newval**) |
| cpp | int **set_callbackUrl**( const string& **newval**) |
| m | -(int) setCallbackUrl : (NSString*) **newval** |
| pas | function **set_callbackUrl**( **newval**: string): integer |
| vb | function **set_callbackUrl**( ByVal **newval** As String) As Integer |
| cs | int **set_callbackUrl**( string **newval**) |
| java | int **set_callbackUrl**( String **newval**) |
| py | def **set_callbackUrl**( **newval**) |
| cmd | YNetwork **target** **set_callbackUrl** **newval** |

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the callback URL to notify significant state changes

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_discoverable() network→setDiscoverable()[network setDiscoverable: ] YNetwork

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

| | |
|---|---|
| js | function **set_discoverable**( **newval**) |
| nodejs | function **set_discoverable**( **newval**) |
| php | function **set_discoverable**( **$newval**) |
| cpp | int **set_discoverable**( Y_DISCOVERABLE_enum **newval**) |
| m | -(int) setDiscoverable : (Y_DISCOVERABLE_enum) **newval** |
| pas | function **set_discoverable**( **newval**: Integer): integer |
| vb | function **set_discoverable**( ByVal **newval** As Integer) As Integer |
| cs | int **set_discoverable**( int **newval**) |
| java | int **set_discoverable**( int **newval**) |
| py | def **set_discoverable**( **newval**) |
| cmd | YNetwork **target** **set_discoverable** **newval** |

**Parameters :**

**newval** either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_logicalName() network→setLogicalName()[network setLogicalName: ] YNetwork

Changes the logical name of the network interface.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YNetwork **target set_logicalName newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the network interface.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## network→set_primaryDNS() network→setPrimaryDNS()[network setPrimaryDNS: ] YNetwork

Changes the IP address of the primary name server to be used by the module.

| | |
|---|---|
| js | function **set_primaryDNS**( **newval**) |
| nodejs | function **set_primaryDNS**( **newval**) |
| php | function **set_primaryDNS**( **$newval**) |
| cpp | int **set_primaryDNS**( const string& **newval**) |
| m | -(int) setPrimaryDNS : (NSString*) **newval** |
| pas | function **set_primaryDNS**( **newval**: string): integer |
| vb | function **set_primaryDNS**( ByVal **newval** As String) As Integer |
| cs | int **set_primaryDNS**( string **newval**) |
| java | int **set_primaryDNS**( String **newval**) |
| py | def **set_primaryDNS**( **newval**) |
| cmd | YNetwork **target** **set_primaryDNS** **newval** |

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

**newval** a string corresponding to the IP address of the primary name server to be used by the module

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_secondaryDNS() network→setSecondaryDNS()[network setSecondaryDNS: ] YNetwork

Changes the IP address of the secondary name server to be used by the module.

| | |
|---|---|
| js | function **set_secondaryDNS**( **newval**) |
| nodejs | function **set_secondaryDNS**( **newval**) |
| php | function **set_secondaryDNS**( **$newval**) |
| cpp | int **set_secondaryDNS**( const string& **newval**) |
| m | -(int) setSecondaryDNS : (NSString*) **newval** |
| pas | function **set_secondaryDNS**( **newval**: string): integer |
| vb | function **set_secondaryDNS**( ByVal **newval** As String) As Integer |
| cs | int **set_secondaryDNS**( string **newval**) |
| java | int **set_secondaryDNS**( String **newval**) |
| py | def **set_secondaryDNS**( **newval**) |
| cmd | YNetwork **target 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 setUserData: ]

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## network→set_userPassword() network→setUserPassword()[network setUserPassword: ] YNetwork

Changes the password for the "user" user.

| | |
|---|---|
| js | function **set_userPassword**( **newval**) |
| nodejs | function **set_userPassword**( **newval**) |
| php | function **set_userPassword**( **$newval**) |
| cpp | int **set_userPassword**( const string& **newval**) |
| m | -(int) setUserPassword : (NSString*) **newval** |
| pas | function **set_userPassword**( **newval**: string): integer |
| vb | function **set_userPassword**( ByVal **newval** As String) As Integer |
| cs | int **set_userPassword**( string **newval**) |
| java | int **set_userPassword**( String **newval**) |
| py | def **set_userPassword**( **newval**) |
| cmd | YNetwork **target** **set_userPassword** **newval** |

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the password for the "user" user

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→set_wwwWatchdogDelay() network→setWwwWatchdogDelay()[network setWwwWatchdogDelay: ] YNetwork

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

| | |
|---|---|
| js | function **set_wwwWatchdogDelay**( **newval**) |
| nodejs | function **set_wwwWatchdogDelay**( **newval**) |
| php | function **set_wwwWatchdogDelay**( **$newval**) |
| cpp | int **set_wwwWatchdogDelay**( int **newval**) |
| m | -(int) setWwwWatchdogDelay : (int) **newval** |
| pas | function **set_wwwWatchdogDelay**( **newval**: LongInt): integer |
| vb | function **set_wwwWatchdogDelay**( ByVal **newval** As Integer) As Integer |
| cs | int **set_wwwWatchdogDelay**( int **newval**) |
| java | int **set_wwwWatchdogDelay**( int **newval**) |
| py | def **set_wwwWatchdogDelay**( **newval**) |
| cmd | YNetwork **target** **set_wwwWatchdogDelay** **newval** |

A zero value disables automated reboot in case of Internet connectivity loss. The smallest valid non-zero timeout is 90 seconds.

**Parameters :**

**newval** an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→useDHCP()[network useDHCP: ] YNetwork

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

```
js      function useDHCP( fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)
nodejs  function useDHCP( fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)
php     function useDHCP( $fallbackIpAddr, $fallbackSubnetMaskLen, $fallbackRouter)
cpp     int useDHCP( string fallbackIpAddr,
                     int fallbackSubnetMaskLen,
                     string fallbackRouter)
m       -(int) useDHCP : (NSString*) fallbackIpAddr
                       : (int) fallbackSubnetMaskLen
                       : (NSString*) fallbackRouter
pas     function useDHCP( fallbackIpAddr: string,
                          fallbackSubnetMaskLen: LongInt,
                          fallbackRouter: string): integer
vb      function useDHCP( ByVal fallbackIpAddr As String,
                          ByVal fallbackSubnetMaskLen As Integer,
                          ByVal fallbackRouter As String) As Integer
cs      int useDHCP( string fallbackIpAddr,
                     int fallbackSubnetMaskLen,
                     string fallbackRouter)
java    int useDHCP( String fallbackIpAddr,
                     int fallbackSubnetMaskLen,
                     String fallbackRouter)
py      def useDHCP( fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)
cmd     YNetwork target useDHCP fallbackIpAddr fallbackSubnetMaskLen fallbackRouter
```

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

| | |
|---|---|
| **fallbackIpAddr** | fallback IP address, to be used when no DHCP reply is received |
| **fallbackSubnetMaskLen** | fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0) |
| **fallbackRouter** | fallback router IP address, to be used when no DHCP reply is received |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→useStaticIP()[network useStaticIP: ] YNetwork

Changes the configuration of the network interface to use a static IP address.

```
js      function useStaticIP( ipAddress, subnetMaskLen, router)
nodejs  function useStaticIP( ipAddress, subnetMaskLen, router)
php     function useStaticIP( $ipAddress, $subnetMaskLen, $router)
cpp     int useStaticIP( string ipAddress,
                         int subnetMaskLen,
                         string router)
m       -(int) useStaticIP : (NSString*) ipAddress
                           : (int) subnetMaskLen
                           : (NSString*) router
pas     function useStaticIP( ipAddress: string,
                              subnetMaskLen: LongInt,
                              router: string): integer
vb      function useStaticIP( ByVal ipAddress As String,
                              ByVal subnetMaskLen As Integer,
                              ByVal router As String) As Integer
cs      int useStaticIP( string ipAddress,
                         int subnetMaskLen,
                         string router)
java    int useStaticIP( String ipAddress,
                         int subnetMaskLen,
                         String router)
py      def useStaticIP( ipAddress, subnetMaskLen, router)
cmd     YNetwork target useStaticIP ipAddress subnetMaskLen router
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

| | |
|---|---|
| **ipAddress** | device IP address |
| **subnetMaskLen** | subnet mask length, as an integer (eg. 24 means 255.255.255.0) |
| **router** | router IP address (default gateway) |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## network→wait_async() YNetwork

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.27. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_oscontrol.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YOsControl = yoctolib.YOsControl; |
| php | require_once('yocto_oscontrol.php'); |
| cpp | #include "yocto_oscontrol.h" |
| m | #import "yocto_oscontrol.h" |
| pas | uses yocto_oscontrol; |
| vb | yocto_oscontrol.vb |
| cs | yocto_oscontrol.cs |
| java | import com.yoctopuce.YoctoAPI.YOsControl; |
| py | from yocto_oscontrol import * |

**Global functions**

**yFindOsControl(func)**

Retrieves OS control for a given identifier.

**yFirstOsControl()**

Starts the enumeration of OS control currently accessible.

**YOsControl methods**

**oscontrol→describe()**

Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**oscontrol→get_advertisedValue()**

Returns the current value of the OS control (no more than 6 characters).

**oscontrol→get_errorMessage()**

Returns the error message of the latest error with the OS control.

**oscontrol→get_errorType()**

Returns the numerical error code of the latest error with the OS control.

**oscontrol→get_friendlyName()**

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

**oscontrol→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**oscontrol→get_functionId()**

Returns the hardware identifier of the OS control, without reference to the module.

**oscontrol→get_hardwareId()**

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

**oscontrol→get_logicalName()**

Returns the logical name of the OS control.

**oscontrol→get_module()**

Gets the `YModule` object for the device on which the function is located.

**oscontrol→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**oscontrol→get_shutdownCountdown()**

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

**oscontrol→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**oscontrol→isOnline()**

Checks if the OS control is currently reachable, without raising any error.

**oscontrol→isOnline_async(callback, context)**

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

**oscontrol→load(msValidity)**

Preloads the OS control cache with a specified validity duration.

**oscontrol→load_async(msValidity, callback, context)**

Preloads the OS control cache with a specified validity duration (asynchronous version).

**oscontrol→nextOsControl()**

Continues the enumeration of OS control started using `yFirstOsControl()`.

**oscontrol→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**oscontrol→set_logicalName(newval)**

Changes the logical name of the OS control.

**oscontrol→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**oscontrol→shutdown(secBeforeShutDown)**

Schedules an OS shutdown after a given number of seconds.

**oscontrol→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YOsControl.FindOsControl() yFindOsControl()yFindOsControl() YOsControl

Retrieves OS control for a given identifier.

| | |
|---|---|
| js | function **yFindOsControl**( **func**) |
| nodejs | function **FindOsControl**( **func**) |
| php | function **yFindOsControl**( **$func**) |
| cpp | YOsControl* **yFindOsControl**( const string& **func**) |
| m | YOsControl* **yFindOsControl**( NSString* **func**) |
| pas | function **yFindOsControl**( **func**: string): TYOsControl |
| vb | function **yFindOsControl**( ByVal **func** As String) As YOsControl |
| cs | YOsControl **FindOsControl**( string **func**) |
| java | YOsControl **FindOsControl**( String **func**) |
| py | def **FindOsControl**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YOsControl.isOnline()` to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the OS control

**Returns :**

a `YOsControl` object allowing you to drive the OS control.

## YOsControl.FirstOsControl() yFirstOsControl()yFirstOsControl() YOsControl

Starts the enumeration of OS control currently accessible.

| | |
|---|---|
| js | function **yFirstOsControl**( ) |
| nodejs | function **FirstOsControl**( ) |
| php | function **yFirstOsControl**( ) |
| cpp | YOsControl* **yFirstOsControl**( ) |
| m | YOsControl* **yFirstOsControl**( ) |
| pas | function **yFirstOsControl**( ): TYOsControl |
| vb | function **yFirstOsControl**( ) As YOsControl |
| cs | YOsControl **FirstOsControl**( ) |
| java | YOsControl **FirstOsControl**( ) |
| py | def **FirstOsControl**( ) |

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

**Returns :**

a pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a `null` pointer if there are none.

## oscontrol→describe()[oscontrol describe] YOsControl

Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the OS control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## oscontrol→get_advertisedValue() oscontrol→advertisedValue()[oscontrol advertisedValue] YOsControl

Returns the current value of the OS control (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YOsControl **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the OS control (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## oscontrol→get_errorMessage() oscontrol→errorMessage()[oscontrol errorMessage] YOsControl

Returns the error message of the latest error with the OS control.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the OS control object

## oscontrol→get_errorType() oscontrol→errorType() YOsControl

Returns the numerical error code of the latest error with the OS control.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the OS control object

## oscontrol→get_friendlyName() oscontrol→friendlyName()[oscontrol friendlyName] YOsControl

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the OS control if they are defined, otherwise the serial number of the module and the hardware identifier of the OS control (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the OS control using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## oscontrol→get_functionDescriptor() oscontrol→functionDescriptor()[oscontrol functionDescriptor] YOsControl

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## oscontrol→get_functionId() oscontrol→functionId()[oscontrol functionId] YOsControl

Returns the hardware identifier of the OS control, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the OS control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## oscontrol→get_hardwareId() oscontrol→hardwareId()[oscontrol hardwareId] YOsControl

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the OS control. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the OS control (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## oscontrol→get_logicalName() oscontrol→logicalName()[oscontrol logicalName] YOsControl

Returns the logical name of the OS control.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YOsControl **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the OS control. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## oscontrol→get_module() oscontrol→module()[oscontrol module] YOsControl

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## oscontrol→get_module_async() oscontrol→module_async() YOsControl

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## oscontrol→get_shutdownCountdown() oscontrol→shutdownCountdown()[oscontrol shutdownCountdown] YOsControl

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

| | |
|---|---|
| js | function **get_shutdownCountdown**( ) |
| nodejs | function **get_shutdownCountdown**( ) |
| php | function **get_shutdownCountdown**( ) |
| cpp | int **get_shutdownCountdown**( ) |
| m | -(int) shutdownCountdown |
| pas | function **get_shutdownCountdown**( ): LongInt |
| vb | function **get_shutdownCountdown**( ) As Integer |
| cs | int **get_shutdownCountdown**( ) |
| java | int **get_shutdownCountdown**( ) |
| py | def **get_shutdownCountdown**( ) |
| cmd | YOsControl **target** **get_shutdownCountdown** |

**Returns :**

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns `Y_SHUTDOWNCOUNTDOWN_INVALID`.

## oscontrol→get_userData() oscontrol→userData()[oscontrol userData] YOsControl

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## oscontrol→isOnline()[oscontrol isOnline] YOsControl

Checks if the OS control is currently reachable, without raising any error.

| | |
|---|---|
| `js` | function **isOnline**( ) |
| `nodejs` | function **isOnline**( ) |
| `php` | function **isOnline**( ) |
| `cpp` | bool **isOnline**( ) |
| `m` | -(BOOL) **isOnline** |
| `pas` | function **isOnline**( ): boolean |
| `vb` | function **isOnline**( ) As Boolean |
| `cs` | bool **isOnline**( ) |
| `java` | boolean **isOnline**( ) |
| `py` | def **isOnline**( ) |

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

**Returns :**
`true` if the OS control can be reached, and `false` otherwise

## oscontrol→isOnline_async() YOsControl

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## oscontrol→load()[oscontrol load: ] YOsControl

Preloads the OS control cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## oscontrol→load_async() YOsControl

Preloads the OS control cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## oscontrol→nextOsControl()[oscontrol nextOsControl] YOsControl

Continues the enumeration of OS control started using `yFirstOsControl()`.

| | |
|---|---|
| js | function **nextOsControl**( ) |
| nodejs | function **nextOsControl**( ) |
| php | function **nextOsControl**( ) |
| cpp | YOsControl * **nextOsControl**( ) |
| m | -(YOsControl*) **nextOsControl** |
| pas | function **nextOsControl**( ): TYOsControl |
| vb | function **nextOsControl**( ) As YOsControl |
| cs | YOsControl **nextOsControl**( ) |
| java | YOsControl **nextOsControl**( ) |
| py | def **nextOsControl**( ) |

**Returns :**

a pointer to a `YOsControl` object, corresponding to OS control currently online, or a `null` pointer if there are no more OS control to enumerate.

## oscontrol→registerValueCallback()[oscontrol registerValueCallback: ] YOsControl

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YOsControlValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YOsControlValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYOsControlValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## oscontrol→set_logicalName() oscontrol→setLogicalName()[oscontrol setLogicalName: ] YOsControl

Changes the logical name of the OS control.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YOsControl **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the OS control.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## oscontrol→set_userData() oscontrol→setUserData()[oscontrol setUserData: ] YOsControl

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## oscontrol→shutdown()[oscontrol shutdown: ] YOsControl

Schedules an OS shutdown after a given number of seconds.

| | |
|---|---|
| js | function **shutdown**( **secBeforeShutDown**) |
| nodejs | function **shutdown**( **secBeforeShutDown**) |
| php | function **shutdown**( **$secBeforeShutDown**) |
| cpp | int **shutdown**( int **secBeforeShutDown**) |
| m | -(int) **shutdown** : (int) **secBeforeShutDown** |
| pas | function **shutdown**( **secBeforeShutDown**: LongInt): LongInt |
| vb | function **shutdown**( ) As Integer |
| cs | int **shutdown**( int **secBeforeShutDown**) |
| java | int **shutdown**( int **secBeforeShutDown**) |
| py | def **shutdown**( **secBeforeShutDown**) |
| cmd | YOsControl **target** **shutdown** **secBeforeShutDown** |

**Parameters :**

**secBeforeShutDown** number of seconds before shutdown

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## oscontrol→wait_async() YOsControl

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

`js` function **wait_async**( **callback**, **context**)
`nodejs` function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.28. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_power.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YPower = yoctolib.YPower; |
| php | require_once('yocto_power.php'); |
| cpp | #include "yocto_power.h" |
| m | #import "yocto_power.h" |
| pas | uses yocto_power; |
| vb | yocto_power.vb |
| cs | yocto_power.cs |
| java | import com.yoctopuce.YoctoAPI.YPower; |
| py | from yocto_power import * |

**Global functions**

**yFindPower(func)**

Retrieves a electrical power sensor for a given identifier.

**yFirstPower()**

Starts the enumeration of electrical power sensors currently accessible.

**YPower methods**

**power→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**power→describe()**

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**power→get_advertisedValue()**

Returns the current value of the electrical power sensor (no more than 6 characters).

**power→get_cosPhi()**

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

**power→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**power→get_currentValue()**

Returns the current measure for the electrical power.

**power→get_errorMessage()**

Returns the error message of the latest error with the electrical power sensor.

**power→get_errorType()**

Returns the numerical error code of the latest error with the electrical power sensor.

**power→get_friendlyName()**

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**power→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**power→get_functionId()**

Returns the hardware identifier of the electrical power sensor, without reference to the module.

**power→get_hardwareId()**

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

**power→get_highestValue()**

Returns the maximal value observed for the electrical power.

**power→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**power→get_logicalName()**

Returns the logical name of the electrical power sensor.

**power→get_lowestValue()**

Returns the minimal value observed for the electrical power.

**power→get_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

**power→get_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

**power→get_module()**

Gets the `YModule` object for the device on which the function is located.

**power→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**power→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**power→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**power→get_resolution()**

Returns the resolution of the measured values.

**power→get_unit()**

Returns the measuring unit for the electrical power.

**power→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**power→isOnline()**

Checks if the electrical power sensor is currently reachable, without raising any error.

**power→isOnline_async(callback, context)**

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

**power→load(msValidity)**

Preloads the electrical power sensor cache with a specified validity duration.

**power→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**power→load_async(msValidity, callback, context)**

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

**power→nextPower()**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

**power→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**power→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**power→reset**()

Resets the energy counter.

**power→set_highestValue**(**newval**)

Changes the recorded maximal value observed pour the electrical power.

**power→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**power→set_logicalName**(**newval**)

Changes the logical name of the electrical power sensor.

**power→set_lowestValue**(**newval**)

Changes the recorded minimal value observed pour the electrical power.

**power→set_reportFrequency**(**newval**)

Changes the timed value notification frequency for this function.

**power→set_resolution**(**newval**)

Changes the resolution of the measured values.

**power→set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

**power→wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YPower.FindPower() yFindPower()yFindPower()

**YPower**

Retrieves a electrical power sensor for a given identifier.

| | |
|---|---|
| js | function **yFindPower**( **func**) |
| nodejs | function **FindPower**( **func**) |
| php | function **yFindPower**( **$func**) |
| cpp | YPower* **yFindPower**( const string& **func**) |
| m | YPower* **yFindPower**( NSString* **func**) |
| pas | function **yFindPower**( **func**: string): TYPower |
| vb | function **yFindPower**( ByVal **func** As String) As YPower |
| cs | YPower **FindPower**( string **func**) |
| java | YPower **FindPower**( String **func**) |
| py | def **FindPower**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the electrical power sensor

**Returns :**

a `YPower` object allowing you to drive the electrical power sensor.

## YPower.FirstPower() yFirstPower()yFirstPower() YPower

Starts the enumeration of electrical power sensors currently accessible.

| | |
|---|---|
| `js` | function **yFirstPower**( ) |
| `nodejs` | function **FirstPower**( ) |
| `php` | function **yFirstPower**( ) |
| `cpp` | YPower* **yFirstPower**( ) |
| `m` | YPower* **yFirstPower**( ) |
| `pas` | function **yFirstPower**( ): TYPower |
| `vb` | function **yFirstPower**( ) As YPower |
| `cs` | YPower **FirstPower**( ) |
| `java` | YPower **FirstPower**( ) |
| `py` | def **FirstPower**( ) |

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

**Returns :**

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

## power→calibrateFromPoints()[power calibrateFromPoints: ] YPower

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YPower **target calibrateFromPoints rawValues refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→describe()[power describe] YPower

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the electrical power sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## power→get_advertisedValue() power→advertisedValue()[power advertisedValue] YPower

Returns the current value of the electrical power sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YPower **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the electrical power sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## power→get_cosPhi() power→cosPhi()[power cosPhi] YPower

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

| | |
|---|---|
| `js` | function **get_cosPhi**( ) |
| `nodejs` | function **get_cosPhi**( ) |
| `php` | function **get_cosPhi**( ) |
| `cpp` | double **get_cosPhi**( ) |
| `m` | -(double) cosPhi |
| `pas` | function **get_cosPhi**( ): double |
| `vb` | function **get_cosPhi**( ) As Double |
| `cs` | double **get_cosPhi**( ) |
| `java` | double **get_cosPhi**( ) |
| `py` | def **get_cosPhi**( ) |
| `cmd` | YPower **target** **get_cosPhi** |

**Returns :**

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns `Y_COSPHI_INVALID`.

## power→get_currentRawValue() power→currentRawValue()[power currentRawValue] YPower

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YPower **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## power→get_currentValue() power→currentValue()[power currentValue] YPower

Returns the current measure for the electrical power.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YPower **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current measure for the electrical power

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## power→get_errorMessage() power→errorMessage()[power errorMessage] YPower

Returns the error message of the latest error with the electrical power sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the electrical power sensor object

## power→get_errorType()
## power→errorType()

**YPower**

Returns the numerical error code of the latest error with the electrical power sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the electrical power sensor object

## power→get_friendlyName() power→friendlyName()[power friendlyName] YPower

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for exemple: `MyCustomName.relay1`)

**Returns :**
a string that uniquely identifies the electrical power sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## power→get_functionDescriptor() power→functionDescriptor()[power functionDescriptor] YPower

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## power→get_functionId() YPower
## power→functionId()[power functionId]

Returns the hardware identifier of the electrical power sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the electrical power sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## power→get_hardwareId() power→hardwareId()[power hardwareId] YPower

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| `js` | function **get_hardwareId**( ) |
| `nodejs` | function **get_hardwareId**( ) |
| `php` | function **get_hardwareId**( ) |
| `cpp` | string **get_hardwareId**( ) |
| `m` | -(NSString*) hardwareId |
| `vb` | function **get_hardwareId**( ) As String |
| `cs` | string **get_hardwareId**( ) |
| `java` | String **get_hardwareId**( ) |
| `py` | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the electrical power sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the electrical power sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## power→get_highestValue()
## power→highestValue()[power highestValue]

YPower

Returns the maximal value observed for the electrical power.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YPower **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the electrical power

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## power→get_logFrequency() power→logFrequency()[power logFrequency] YPower

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YPower **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## power→get_logicalName() power→logicalName()[power logicalName] YPower

Returns the logical name of the electrical power sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YPower **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the electrical power sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## power→get_lowestValue() power→lowestValue()[power lowestValue] YPower

Returns the minimal value observed for the electrical power.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YPower **target** **get_lowestValue** |

**Returns :**

a floating point number corresponding to the minimal value observed for the electrical power

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## power→get_meter() power→meter()[power meter] YPower

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

| | |
|---|---|
| js | function **get_meter**( ) |
| nodejs | function **get_meter**( ) |
| php | function **get_meter**( ) |
| cpp | double **get_meter**( ) |
| m | -(double) meter |
| pas | function **get_meter**( ): double |
| vb | function **get_meter**( ) As Double |
| cs | double **get_meter**( ) |
| java | double **get_meter**( ) |
| py | def **get_meter**( ) |
| cmd | YPower **target** **get_meter** |

Note that this counter is reset at each start of the device.

**Returns :**
a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns `Y_METER_INVALID`.

## power→get_meterTimer() power→meterTimer()[power meterTimer] YPower

Returns the elapsed time since last energy counter reset, in seconds.

| | |
|---|---|
| js | function **get_meterTimer**( ) |
| nodejs | function **get_meterTimer**( ) |
| php | function **get_meterTimer**( ) |
| cpp | int **get_meterTimer**( ) |
| m | -(int) meterTimer |
| pas | function **get_meterTimer**( ): LongInt |
| vb | function **get_meterTimer**( ) As Integer |
| cs | int **get_meterTimer**( ) |
| java | int **get_meterTimer**( ) |
| py | def **get_meterTimer**( ) |
| cmd | YPower **target** **get_meterTimer** |

**Returns :**
an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns `Y_METERTIMER_INVALID`.

## power→get_module() power→module()[power module] YPower

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## power→get_module_async() power→module_async()

**YPower**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## power→get_recordedData() YPower
## power→recordedData()[power recordedData: ]

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime** : (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YPower **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## power→get_reportFrequency() power→reportFrequency()[power reportFrequency] YPower

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YPower **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## power→get_resolution() power→resolution()[power resolution] YPower

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YPower **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## power→get_unit() power→unit()[power unit]

**YPower**

Returns the measuring unit for the electrical power.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YPower **target** **get_unit** |

**Returns :**

a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## power→get_userData() power→userData()[power userData] YPower

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## power→isOnline()[power isOnline] YPower

Checks if the electrical power sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

**Returns :**
`true` if the electrical power sensor can be reached, and `false` otherwise

## power→isOnline_async() YPower

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## power→load()[power load: ] YPower

Preloads the electrical power sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## power→loadCalibrationPoints()[power loadCalibrationPoints: ] YPower

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YPower **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→load_async() YPower

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## power→nextPower()[power nextPower] YPower

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

| | |
|---|---|
| js | function **nextPower**( ) |
| nodejs | function **nextPower**( ) |
| php | function **nextPower**( ) |
| cpp | YPower * **nextPower**( ) |
| m | -(YPower*) **nextPower** |
| pas | function **nextPower**( ): TYPower |
| vb | function **nextPower**( ) As YPower |
| cs | YPower **nextPower**( ) |
| java | YPower **nextPower**( ) |
| py | def **nextPower**( ) |

**Returns :**

a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a `null` pointer if there are no more electrical power sensors to enumerate.

## power→registerTimedReportCallback()[power registerTimedReportCallback: ] YPower

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YPowerTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YPowerTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYPowerTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## power→registerValueCallback()[power registerValueCallback: ] YPower

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YPowerValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YPowerValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYPowerValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## power→reset()[power reset] YPower

Resets the energy counter.

| | |
|---|---|
| js | function **reset**( ) |
| nodejs | function **reset**( ) |
| php | function **reset**( ) |
| cpp | int **reset**( ) |
| m | -(int) **reset** |
| pas | function **reset**( ): LongInt |
| vb | function **reset**( ) As Integer |
| cs | int **reset**( ) |
| java | int **reset**( ) |
| py | def **reset**( ) |
| cmd | YPower **target** **reset** |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→set_highestValue() power→setHighestValue()[power setHighestValue: ] YPower

Changes the recorded maximal value observed pour the electrical power.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YPower **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed pour the electrical power

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→set_logFrequency() power→setLogFrequency()[power setLogFrequency: ] YPower

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YPower **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→set_logicalName() power→setLogicalName()[power setLogicalName: ] YPower

Changes the logical name of the electrical power sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YPower **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the electrical power sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## power→set_lowestValue() power→setLowestValue()[power setLowestValue: ] YPower

Changes the recorded minimal value observed pour the electrical power.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YPower **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed pour the electrical power

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→set_reportFrequency() power→setReportFrequency()[power setReportFrequency: ] YPower

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YPower **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→set_resolution() power→setResolution()[power setResolution: ] YPower

Changes the resolution of the measured values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YPower **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## power→set_userData() power→setUserData()[power setUserData: ] YPower

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## power→wait_async() YPower

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.29. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_pressure.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YPressure = yoctolib.YPressure; |
| php | require_once('yocto_pressure.php'); |
| cpp | #include "yocto_pressure.h" |
| m | #import "yocto_pressure.h" |
| pas | uses yocto_pressure; |
| vb | yocto_pressure.vb |
| cs | yocto_pressure.cs |
| java | import com.yoctopuce.YoctoAPI.YPressure; |
| py | from yocto_pressure import * |

**Global functions**

**yFindPressure(func)**

Retrieves a pressure sensor for a given identifier.

**yFirstPressure()**

Starts the enumeration of pressure sensors currently accessible.

**`YPressure` methods**

**pressure→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**pressure→describe()**

Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**pressure→get_advertisedValue()**

Returns the current value of the pressure sensor (no more than 6 characters).

**pressure→get_currentRawValue()**

Returns the unrounded and uncalibrated raw value returned by the sensor.

**pressure→get_currentValue()**

Returns the current measure for the pressure.

**pressure→get_errorMessage()**

Returns the error message of the latest error with the pressure sensor.

**pressure→get_errorType()**

Returns the numerical error code of the latest error with the pressure sensor.

**pressure→get_friendlyName()**

Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**pressure→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**pressure→get_functionId()**

Returns the hardware identifier of the pressure sensor, without reference to the module.

**pressure→get_hardwareId()**

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

**pressure→get_highestValue()**
Returns the maximal value observed for the pressure.

**pressure→get_logFrequency()**
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**pressure→get_logicalName()**
Returns the logical name of the pressure sensor.

**pressure→get_lowestValue()**
Returns the minimal value observed for the pressure.

**pressure→get_module()**
Gets the `YModule` object for the device on which the function is located.

**pressure→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**pressure→get_recordedData(startTime, endTime)**
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**pressure→get_reportFrequency()**
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**pressure→get_resolution()**
Returns the resolution of the measured values.

**pressure→get_unit()**
Returns the measuring unit for the pressure.

**pressure→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**pressure→isOnline()**
Checks if the pressure sensor is currently reachable, without raising any error.

**pressure→isOnline_async(callback, context)**
Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

**pressure→load(msValidity)**
Preloads the pressure sensor cache with a specified validity duration.

**pressure→loadCalibrationPoints(rawValues, refValues)**
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**pressure→load_async(msValidity, callback, context)**
Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

**pressure→nextPressure()**
Continues the enumeration of pressure sensors started using `yFirstPressure()`.

**pressure→registerTimedReportCallback(callback)**
Registers the callback function that is invoked on every periodic timed notification.

**pressure→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**pressure→set_highestValue(newval)**
Changes the recorded maximal value observed for the pressure.

**pressure→set_logFrequency(newval)**
Changes the datalogger recording frequency for this function.

**pressure→set_logicalName(newval)**
Changes the logical name of the pressure sensor.

**pressure→set_lowestValue(newval)**

Changes the recorded minimal value observed for the pressure.

**pressure→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**pressure→set_resolution(newval)**

Changes the resolution of the measured physical values.

**pressure→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**pressure→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YPressure.FindPressure() yFindPressure()yFindPressure() YPressure

Retrieves a pressure sensor for a given identifier.

| | |
|---|---|
| js | function **yFindPressure**( **func**) |
| nodejs | function **FindPressure**( **func**) |
| php | function **yFindPressure**( **$func**) |
| cpp | YPressure* **yFindPressure**( const string& **func**) |
| m | YPressure* **yFindPressure**( NSString* **func**) |
| pas | function **yFindPressure**( **func**: string): TYPressure |
| vb | function **yFindPressure**( ByVal **func** As String) As YPressure |
| cs | YPressure **FindPressure**( string **func**) |
| java | YPressure **FindPressure**( String **func**) |
| py | def **FindPressure**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the pressure sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPressure.isOnline()` to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the pressure sensor

**Returns :**

a `YPressure` object allowing you to drive the pressure sensor.

## YPressure.FirstPressure() yFirstPressure()yFirstPressure() YPressure

Starts the enumeration of pressure sensors currently accessible.

| | |
|---|---|
| js | function **yFirstPressure**( ) |
| nodejs | function **FirstPressure**( ) |
| php | function **yFirstPressure**( ) |
| cpp | YPressure* **yFirstPressure**( ) |
| m | YPressure* **yFirstPressure**( ) |
| pas | function **yFirstPressure**( ): TYPressure |
| vb | function **yFirstPressure**( ) As YPressure |
| cs | YPressure **FirstPressure**( ) |
| java | YPressure **FirstPressure**( ) |
| py | def **FirstPressure**( ) |

Use the method `YPressure.nextPressure()` to iterate on next pressure sensors.

**Returns :**

a pointer to a `YPressure` object, corresponding to the first pressure sensor currently online, or a `null` pointer if there are none.

## pressure→calibrateFromPoints()[pressure calibrateFromPoints: ] YPressure

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YPressure **target calibrateFromPoints rawValues refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→describe()[pressure describe] YPressure

Returns a short text that describes unambiguously the instance of the pressure sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the pressure sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## pressure→get_advertisedValue() pressure→advertisedValue()[pressure advertisedValue] YPressure

Returns the current value of the pressure sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YPressure **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the pressure sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## pressure→get_currentRawValue() pressure→currentRawValue()[pressure currentRawValue] YPressure

Returns the unrounded and uncalibrated raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YPressure **target** **get_currentRawValue** |

**Returns :**
a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## pressure→get_currentValue() pressure→currentValue()[pressure currentValue] YPressure

Returns the current measure for the pressure.

| | |
|---|---|
| `js` | function **get_currentValue**( ) |
| `nodejs` | function **get_currentValue**( ) |
| `php` | function **get_currentValue**( ) |
| `cpp` | double **get_currentValue**( ) |
| `m` | -(double) currentValue |
| `pas` | function **get_currentValue**( ): double |
| `vb` | function **get_currentValue**( ) As Double |
| `cs` | double **get_currentValue**( ) |
| `java` | double **get_currentValue**( ) |
| `py` | def **get_currentValue**( ) |
| `cmd` | YPressure **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current measure for the pressure

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## pressure→get_errorMessage() pressure→errorMessage()[pressure errorMessage] YPressure

Returns the error message of the latest error with the pressure sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the pressure sensor object

## pressure→get_errorType() pressure→errorType() YPressure

Returns the numerical error code of the latest error with the pressure sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the pressure sensor object

## pressure→get_friendlyName() pressure→friendlyName()[pressure friendlyName] YPressure

Returns a global identifier of the pressure sensor in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the pressure sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the pressure sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the pressure sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## pressure→get_functionDescriptor() pressure→functionDescriptor()[pressure functionDescriptor] YPressure

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## pressure→get_functionId() YPressure
## pressure→functionId()[pressure functionId]

Returns the hardware identifier of the pressure sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the pressure sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## pressure→get_hardwareId() pressure→hardwareId()[pressure hardwareId] YPressure

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the pressure sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the pressure sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## pressure→get_highestValue() pressure→highestValue()[pressure highestValue] YPressure

Returns the maximal value observed for the pressure.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YPressure **target** **get_highestValue** |

**Returns :**
a floating point number corresponding to the maximal value observed for the pressure

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

### pressure→get_logFrequency() pressure→logFrequency()[pressure logFrequency] YPressure

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YPressure **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## pressure→get_logicalName() pressure→logicalName()[pressure logicalName] YPressure

Returns the logical name of the pressure sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YPressure **target get_logicalName** |

**Returns :**

a string corresponding to the logical name of the pressure sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## pressure→get_lowestValue() pressure→lowestValue()[pressure lowestValue] YPressure

Returns the minimal value observed for the pressure.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YPressure **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the pressure

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## pressure→get_module() pressure→module()[pressure module] YPressure

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## pressure→get_module_async() pressure→module_async() YPressure

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| `js` | function **get_module_async**( **callback**, **context**) |
| `nodejs` | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pressure→get_recordedData() YPressure
## pressure→recordedData()[pressure recordedData: ]

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YPressure **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## pressure→get_reportFrequency() pressure→reportFrequency()[pressure reportFrequency] YPressure

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YPressure **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## pressure→get_resolution() pressure→resolution()[pressure resolution] YPressure

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YPressure **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## pressure→get_unit() pressure→unit()[pressure unit] YPressure

Returns the measuring unit for the pressure.

| | |
|---|---|
| `js` | function **get_unit**( ) |
| `nodejs` | function **get_unit**( ) |
| `php` | function **get_unit**( ) |
| `cpp` | string **get_unit**( ) |
| `m` | -(NSString*) unit |
| `pas` | function **get_unit**( ): string |
| `vb` | function **get_unit**( ) As String |
| `cs` | string **get_unit**( ) |
| `java` | String **get_unit**( ) |
| `py` | def **get_unit**( ) |
| `cmd` | YPressure **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## pressure→get_userData() pressure→userData()[pressure userData] YPressure

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## pressure→isOnline()[pressure isOnline] YPressure

Checks if the pressure sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

**Returns :**

`true` if the pressure sensor can be reached, and `false` otherwise

## pressure→isOnline_async() YPressure

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pressure→load()[pressure load: ] YPressure

Preloads the pressure sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## pressure→loadCalibrationPoints()[pressure loadCalibrationPoints: ] YPressure

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YPressure **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→load_async() YPressure

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pressure→nextPressure()[pressure nextPressure] YPressure

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

| | |
|---|---|
| js | function **nextPressure**( ) |
| nodejs | function **nextPressure**( ) |
| php | function **nextPressure**( ) |
| cpp | YPressure * **nextPressure**( ) |
| m | -(YPressure*) **nextPressure** |
| pas | function **nextPressure**( ): TYPressure |
| vb | function **nextPressure**( ) As YPressure |
| cs | YPressure **nextPressure**( ) |
| java | YPressure **nextPressure**( ) |
| py | def **nextPressure**( ) |

**Returns :**

a pointer to a `YPressure` object, corresponding to a pressure sensor currently online, or a `null` pointer if there are no more pressure sensors to enumerate.

## pressure→registerTimedReportCallback()[pressure registerTimedReportCallback: ] YPressure

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YPressureTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YPressureTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYPressureTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## pressure→registerValueCallback()[pressure registerValueCallback: ] YPressure

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YPressureValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YPressureValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYPressureValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## pressure→set_highestValue() pressure→setHighestValue()[pressure setHighestValue: ] YPressure

Changes the recorded maximal value observed for the pressure.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YPressure **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the pressure

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→set_logFrequency() pressure→setLogFrequency()[pressure setLogFrequency: ] YPressure

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YPressure **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→set_logicalName() pressure→setLogicalName()[pressure setLogicalName: ] YPressure

Changes the logical name of the pressure sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YPressure **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the pressure sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## pressure→set_lowestValue() pressure→setLowestValue()[pressure setLowestValue: ] YPressure

Changes the recorded minimal value observed for the pressure.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YPressure **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the pressure

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→set_reportFrequency() pressure→setReportFrequency()[pressure setReportFrequency: ] YPressure

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YPressure **target set_reportFrequency newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→set_resolution() pressure→setResolution()[pressure setResolution: ] YPressure

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YPressure **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pressure→set_userData() pressure→setUserData()[pressure setUserData: ] YPressure

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## pressure→wait_async() YPressure

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.30. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_pwmoutput.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YPwmOutput = yoctolib.YPwmOutput; |
| php | require_once('yocto_pwmoutput.php'); |
| cpp | #include "yocto_pwmoutput.h" |
| m | #import "yocto_pwmoutput.h" |
| pas | uses yocto_pwmoutput; |
| vb | yocto_pwmoutput.vb |
| cs | yocto_pwmoutput.cs |
| java | import com.yoctopuce.YoctoAPI.YPwmOutput; |
| py | from yocto_pwmoutput import * |

**Global functions**

**yFindPwmOutput(func)**
Retrieves a PWM for a given identifier.

**yFirstPwmOutput()**
Starts the enumeration of PWMs currently accessible.

**`YPwmOutput` methods**

**pwmoutput→describe()**
Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**pwmoutput→dutyCycleMove(target, ms_duration)**
Performs a smooth change of the pulse duration toward a given value.

**pwmoutput→get_advertisedValue()**
Returns the current value of the PWM (no more than 6 characters).

**pwmoutput→get_dutyCycle()**
Returns the PWMs dutty cyle as a floating point number between 0 an 1.

**pwmoutput→get_dutyCycleAtPowerOn()**
Returns the PWMs duty cycle at device power up as a floating point number between 0.0 and 100.

**pwmoutput→get_enabled()**
Returns the state of the PWMs.

**pwmoutput→get_enabledAtPowerOn()**
Returns the state of the PWMs at device power up.

**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 nanoseconde.

**pwmoutput→get_pulseDuration()**

Returns the PWM pulse length in milliseconds.

**pwmoutput→get_userData()**

Returns the value of the userData attribute, as previously stored using method set_userData.

**pwmoutput→isOnline()**

Checks if the PWM is currently reachable, without raising any error.

**pwmoutput→isOnline_async(callback, context)**

Checks if the PWM is currently reachable, without raising any error (asynchronous version).

**pwmoutput→load(msValidity)**

Preloads the PWM cache with a specified validity duration.

**pwmoutput→load_async(msValidity, callback, context)**

Preloads the PWM cache with a specified validity duration (asynchronous version).

**pwmoutput→nextPwmOutput()**

Continues the enumeration of PWMs started using yFirstPwmOutput().

**pwmoutput→pulseDurationMove(ms_target, ms_duration)**

Performs a smooth change 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)**

Configures the PWMs duty cyle.

**pwmoutput→set_dutyCycleAtPowerOn(newval)**

Configures the PWMs duty cycle at device power up.

**pwmoutput→set_enabled(newval)**

Stops or starts the PWM.

**pwmoutput→set_enabledAtPowerOn(newval)**

Configures the state of PWM at device power up.

**pwmoutput→set_frequency(newval)**

Configures the PWM frequency.

**pwmoutput→set_logicalName(newval)**

Changes the logical name of the PWM.

**pwmoutput→set_period(newval)**

Configures the PWM period.

**pwmoutput→set_pulseDuration(newval)**

Configures the PWM pluses length.

**pwmoutput→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**pwmoutput→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YPwmOutput.FindPwmOutput() yFindPwmOutput()yFindPwmOutput()   YPwmOutput

Retrieves a PWM for a given identifier.

| | |
|---|---|
| js | function **yFindPwmOutput**( **func**) |
| nodejs | function **FindPwmOutput**( **func**) |
| php | function **yFindPwmOutput**( **$func**) |
| cpp | YPwmOutput* **yFindPwmOutput**( const string& **func**) |
| m | YPwmOutput* **yFindPwmOutput**( NSString* **func**) |
| pas | function **yFindPwmOutput**( **func**: string): TYPwmOutput |
| vb | function **yFindPwmOutput**( ByVal **func** As String) As YPwmOutput |
| cs | YPwmOutput **FindPwmOutput**( string **func**) |
| java | YPwmOutput **FindPwmOutput**( String **func**) |
| py | def **FindPwmOutput**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the PWM

**Returns :**

a `YPwmOutput` object allowing you to drive the PWM.

## YPwmOutput.FirstPwmOutput() yFirstPwmOutput()yFirstPwmOutput() YPwmOutput

Starts the enumeration of PWMs currently accessible.

| | |
|---|---|
| js | function **yFirstPwmOutput**( ) |
| nodejs | function **FirstPwmOutput**( ) |
| php | function **yFirstPwmOutput**( ) |
| cpp | YPwmOutput* **yFirstPwmOutput**( ) |
| m | YPwmOutput* **yFirstPwmOutput**( ) |
| pas | function **yFirstPwmOutput**( ): TYPwmOutput |
| vb | function **yFirstPwmOutput**( ) As YPwmOutput |
| cs | YPwmOutput **FirstPwmOutput**( ) |
| java | YPwmOutput **FirstPwmOutput**( ) |
| py | def **FirstPwmOutput**( ) |

Use the method `YPwmOutput.nextPwmOutput()` to iterate on next PWMs.

**Returns :**

a pointer to a `YPwmOutput` object, corresponding to the first PWM currently online, or a `null` pointer if there are none.

## pwmoutput→describe()[pwmoutput describe] YPwmOutput

Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the PWM (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## pwmoutput→dutyCycleMove()[pwmoutput dutyCycleMove: ] YPwmOutput

Performs a smooth change of the pulse duration toward a given value.

| | |
|---|---|
| js | function **dutyCycleMove**( **target**, **ms_duration**) |
| nodejs | function **dutyCycleMove**( **target**, **ms_duration**) |
| php | function **dutyCycleMove**( **$target**, **$ms_duration**) |
| cpp | int **dutyCycleMove**( double **target**, int **ms_duration**) |
| m | -(int) **dutyCycleMove** : (double) **target** : (int) **ms_duration** |
| pas | function **dutyCycleMove**( **target**: double, **ms_duration**: LongInt): LongInt |
| vb | function **dutyCycleMove**( ) As Integer |
| cs | int **dutyCycleMove**( double **target**, int **ms_duration**) |
| java | int **dutyCycleMove**( double **target**, int **ms_duration**) |
| py | def **dutyCycleMove**( **target**, **ms_duration**) |
| cmd | YPwmOutput **target** **dutyCycleMove** **target** **ms_duration** |

**Parameters :**

| | |
|---|---|
| **target** | new duty cycle at the end of the transition (floating-point number, between 0 and 1) |
| **ms_duration** | total duration of the transition, in milliseconds |

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## pwmoutput→get_advertisedValue() pwmoutput→advertisedValue()[pwmoutput advertisedValue] YPwmOutput

Returns the current value of the PWM (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YPwmOutput **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the PWM (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## pwmoutput→get_dutyCycle() pwmoutput→dutyCycle()[pwmoutput dutyCycle] YPwmOutput

Returns the PWMs dutty cyle as a floating point number between 0 an 1.

| | |
|---|---|
| js | function **get_dutyCycle**( ) |
| nodejs | function **get_dutyCycle**( ) |
| php | function **get_dutyCycle**( ) |
| cpp | double **get_dutyCycle**( ) |
| m | -(double) dutyCycle |
| pas | function **get_dutyCycle**( ): double |
| vb | function **get_dutyCycle**( ) As Double |
| cs | double **get_dutyCycle**( ) |
| java | double **get_dutyCycle**( ) |
| py | def **get_dutyCycle**( ) |
| cmd | YPwmOutput **target** **get_dutyCycle** |

**Returns :**
a floating point number corresponding to the PWMs dutty cyle as a floating point number between 0 an 1

On failure, throws an exception or returns `Y_DUTYCYCLE_INVALID`.

## pwmoutput→get_dutyCycleAtPowerOn() pwmoutput→dutyCycleAtPowerOn()[pwmoutput dutyCycleAtPowerOn] YPwmOutput

Returns the PWMs duty cycle at device power up as a floating point number between 0.0 and 100.

| | |
|---|---|
| js | function **get_dutyCycleAtPowerOn**( ) |
| nodejs | function **get_dutyCycleAtPowerOn**( ) |
| php | function **get_dutyCycleAtPowerOn**( ) |
| cpp | double **get_dutyCycleAtPowerOn**( ) |
| m | -(double) dutyCycleAtPowerOn |
| pas | function **get_dutyCycleAtPowerOn**( ): double |
| vb | function **get_dutyCycleAtPowerOn**( ) As Double |
| cs | double **get_dutyCycleAtPowerOn**( ) |
| java | double **get_dutyCycleAtPowerOn**( ) |
| py | def **get_dutyCycleAtPowerOn**( ) |
| cmd | YPwmOutput **target** **get_dutyCycleAtPowerOn** |

0%

**Returns :**

a floating point number corresponding to the PWMs duty cycle at device power up as a floating point number between 0.0 and 100

On failure, throws an exception or returns `Y_DUTYCYCLEATPOWERON_INVALID`.

## pwmoutput→get_enabled() pwmoutput→enabled()[pwmoutput enabled] YPwmOutput

Returns the state of the PWMs.

| | |
|---|---|
| js | function **get_enabled**( ) |
| nodejs | function **get_enabled**( ) |
| php | function **get_enabled**( ) |
| cpp | Y_ENABLED_enum **get_enabled**( ) |
| m | -(Y_ENABLED_enum) enabled |
| pas | function **get_enabled**( ): Integer |
| vb | function **get_enabled**( ) As Integer |
| cs | int **get_enabled**( ) |
| java | int **get_enabled**( ) |
| py | def **get_enabled**( ) |
| cmd | YPwmOutput **target** **get_enabled** |

**Returns :**

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the state of the PWMs

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

## pwmoutput→get_enabledAtPowerOn() pwmoutput→enabledAtPowerOn()[pwmoutput enabledAtPowerOn] YPwmOutput

Returns the state of the PWMs at device power up.

| | |
|---|---|
| js | function **get_enabledAtPowerOn**( ) |
| nodejs | function **get_enabledAtPowerOn**( ) |
| php | function **get_enabledAtPowerOn**( ) |
| cpp | Y_ENABLEDATPOWERON_enum **get_enabledAtPowerOn**( ) |
| m | -(Y_ENABLEDATPOWERON_enum) enabledAtPowerOn |
| pas | function **get_enabledAtPowerOn**( ): Integer |
| vb | function **get_enabledAtPowerOn**( ) As Integer |
| cs | int **get_enabledAtPowerOn**( ) |
| java | int **get_enabledAtPowerOn**( ) |
| py | def **get_enabledAtPowerOn**( ) |
| cmd | YPwmOutput **target** **get_enabledAtPowerOn** |

**Returns :**

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWMs at device power up

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

## pwmoutput→get_errorMessage() pwmoutput→errorMessage()[pwmoutput errorMessage] YPwmOutput

Returns the error message of the latest error with the PWM.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the PWM object

## pwmoutput→get_errorType() pwmoutput→errorType() YPwmOutput

Returns the numerical error code of the latest error with the PWM.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the PWM object

## pwmoutput→get_frequency() pwmoutput→frequency()[pwmoutput frequency] YPwmOutput

Returns the PWM frequency in Hz.

| | |
|---|---|
| js | function **get_frequency**( ) |
| nodejs | function **get_frequency**( ) |
| php | function **get_frequency**( ) |
| cpp | int **get_frequency**( ) |
| m | -(int) frequency |
| pas | function **get_frequency**( ): LongInt |
| vb | function **get_frequency**( ) As Integer |
| cs | int **get_frequency**( ) |
| java | int **get_frequency**( ) |
| py | def **get_frequency**( ) |
| cmd | YPwmOutput **target** **get_frequency** |

**Returns :**
an integer corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`.

## pwmoutput→get_friendlyName() pwmoutput→friendlyName()[pwmoutput friendlyName] YPwmOutput

Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the PWM if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the PWM using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## pwmoutput→get_functionDescriptor() pwmoutput→functionDescriptor()[pwmoutput functionDescriptor] YPwmOutput

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## pwmoutput→get_functionId() pwmoutput→functionId()[pwmoutput functionId] YPwmOutput

Returns the hardware identifier of the PWM, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the PWM (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## pwmoutput→get_hardwareId() pwmoutput→hardwareId()[pwmoutput hardwareId] YPwmOutput

Returns the unique hardware identifier of the PWM in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the PWM (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## pwmoutput→get_logicalName() YPwmOutput
## pwmoutput→logicalName()[pwmoutput logicalName]

Returns the logical name of the PWM.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YPwmOutput **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the PWM. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## pwmoutput→get_module() pwmoutput→module()[pwmoutput module] YPwmOutput

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## pwmoutput→get_module_async() pwmoutput→module_async() YPwmOutput

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pwmoutput→get_period() pwmoutput→period()[pwmoutput period] YPwmOutput

Returns the PWM period in nonaseconde.

| | |
|---|---|
| js | function **get_period**( ) |
| nodejs | function **get_period**( ) |
| php | function **get_period**( ) |
| cpp | double **get_period**( ) |
| m | -(double) period |
| pas | function **get_period**( ): double |
| vb | function **get_period**( ) As Double |
| cs | double **get_period**( ) |
| java | double **get_period**( ) |
| py | def **get_period**( ) |
| cmd | YPwmOutput **target** **get_period** |

**Returns :**

a floating point number corresponding to the PWM period in nonaseconde

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

## pwmoutput→get_pulseDuration() pwmoutput→pulseDuration()[pwmoutput pulseDuration] YPwmOutput

Returns the PWM pulse length in milliseconds.

| | |
|---|---|
| js | function **get_pulseDuration**( ) |
| nodejs | function **get_pulseDuration**( ) |
| php | function **get_pulseDuration**( ) |
| cpp | double **get_pulseDuration**( ) |
| m | -(double) pulseDuration |
| pas | function **get_pulseDuration**( ): double |
| vb | function **get_pulseDuration**( ) As Double |
| cs | double **get_pulseDuration**( ) |
| java | double **get_pulseDuration**( ) |
| py | def **get_pulseDuration**( ) |
| cmd | YPwmOutput **target** **get_pulseDuration** |

**Returns :**
a floating point number corresponding to the PWM pulse length in milliseconds

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`.

## pwmoutput→get_userData() pwmoutput→userData()[pwmoutput userData] YPwmOutput

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## pwmoutput→isOnline()[pwmoutput isOnline] YPwmOutput

Checks if the PWM is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

**Returns :**
`true` if the PWM can be reached, and `false` otherwise

## pwmoutput→isOnline_async() YPwmOutput

Checks if the PWM is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pwmoutput→load()[pwmoutput load: ] YPwmOutput

Preloads the PWM cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## pwmoutput→load_async() YPwmOutput

Preloads the PWM cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pwmoutput→nextPwmOutput()[pwmoutput nextPwmOutput] YPwmOutput

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

| | |
|---|---|
| js | function **nextPwmOutput**( ) |
| nodejs | function **nextPwmOutput**( ) |
| php | function **nextPwmOutput**( ) |
| cpp | YPwmOutput * **nextPwmOutput**( ) |
| m | -(YPwmOutput*) **nextPwmOutput** |
| pas | function **nextPwmOutput**( ): TYPwmOutput |
| vb | function **nextPwmOutput**( ) As YPwmOutput |
| cs | YPwmOutput **nextPwmOutput**( ) |
| java | YPwmOutput **nextPwmOutput**( ) |
| py | def **nextPwmOutput**( ) |

**Returns :**

a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.

## pwmoutput→pulseDurationMove()[pwmoutput pulseDurationMove: ] YPwmOutput

Performs a smooth change of the pulse duration toward a given value.

| | |
|---|---|
| js | function **pulseDurationMove**( **ms_target**, **ms_duration**) |
| nodejs | function **pulseDurationMove**( **ms_target**, **ms_duration**) |
| php | function **pulseDurationMove**( **$ms_target**, **$ms_duration**) |
| cpp | int **pulseDurationMove**( double **ms_target**, int **ms_duration**) |
| m | -(int) **pulseDurationMove** : (double) **ms_target** : (int) **ms_duration** |
| pas | function **pulseDurationMove**( **ms_target**: double, **ms_duration**: LongInt): LongInt |
| vb | function **pulseDurationMove**( ) As Integer |
| cs | int **pulseDurationMove**( double **ms_target**, int **ms_duration**) |
| java | int **pulseDurationMove**( double **ms_target**, int **ms_duration**) |
| py | def **pulseDurationMove**( **ms_target**, **ms_duration**) |
| cmd | YPwmOutput **target** **pulseDurationMove** **ms_target** **ms_duration** |

**Parameters :**

**ms_target** new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)

**ms_duration** total duration of the transition, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## pwmoutput→registerValueCallback()[pwmoutput registerValueCallback: ] YPwmOutput

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YPwmOutputValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YPwmOutputValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYPwmOutputValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## pwmoutput→set_dutyCycle() pwmoutput→setDutyCycle()[pwmoutput setDutyCycle: ] YPwmOutput

Configures the PWMs duty cyle.

| | |
|---|---|
| js | function **set_dutyCycle**( **newval**) |
| nodejs | function **set_dutyCycle**( **newval**) |
| php | function **set_dutyCycle**( **$newval**) |
| cpp | int **set_dutyCycle**( double **newval**) |
| m | -(int) setDutyCycle : (double) **newval** |
| pas | function **set_dutyCycle**( **newval**: double): integer |
| vb | function **set_dutyCycle**( ByVal **newval** As Double) As Integer |
| cs | int **set_dutyCycle**( double **newval**) |
| java | int **set_dutyCycle**( double **newval**) |
| py | def **set_dutyCycle**( **newval**) |
| cmd | YPwmOutput **target** **set_dutyCycle** **newval** |

**Parameters :**

**newval** a floating point number

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmoutput→set_dutyCycleAtPowerOn() pwmoutput→setDutyCycleAtPowerOn()[pwmoutput setDutyCycleAtPowerOn: ] YPwmOutput

Configures the PWMs duty cycle at device power up.

| | |
|---|---|
| js | function **set_dutyCycleAtPowerOn**( **newval**) |
| nodejs | function **set_dutyCycleAtPowerOn**( **newval**) |
| php | function **set_dutyCycleAtPowerOn**( **$newval**) |
| cpp | int **set_dutyCycleAtPowerOn**( double **newval**) |
| m | -(int) setDutyCycleAtPowerOn : (double) **newval** |
| pas | function **set_dutyCycleAtPowerOn**( **newval**: double): integer |
| vb | function **set_dutyCycleAtPowerOn**( ByVal **newval** As Double) As Integer |
| cs | int **set_dutyCycleAtPowerOn**( double **newval**) |
| java | int **set_dutyCycleAtPowerOn**( double **newval**) |
| py | def **set_dutyCycleAtPowerOn**( **newval**) |
| cmd | YPwmOutput **target** **set_dutyCycleAtPowerOn** **newval** |

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** a floating point number

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmoutput→set_enabled() pwmoutput→setEnabled()[pwmoutput setEnabled: ] YPwmOutput

Stops or starts the PWM.

| | |
|---|---|
| js | function **set_enabled**( **newval**) |
| nodejs | function **set_enabled**( **newval**) |
| php | function **set_enabled**( **$newval**) |
| cpp | int **set_enabled**( Y_ENABLED_enum **newval**) |
| m | -(int) setEnabled : (Y_ENABLED_enum) **newval** |
| pas | function **set_enabled**( **newval**: Integer): integer |
| vb | function **set_enabled**( ByVal **newval** As Integer) As Integer |
| cs | int **set_enabled**( int **newval**) |
| java | int **set_enabled**( int **newval**) |
| py | def **set_enabled**( **newval**) |
| cmd | YPwmOutput **target** **set_enabled** **newval** |

**Parameters :**

**newval** either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmoutput→set_enabledAtPowerOn() pwmoutput→setEnabledAtPowerOn()[pwmoutput setEnabledAtPowerOn: ] YPwmOutput

Configures the state of PWM at device power up.

| | |
|---|---|
| js | function **set_enabledAtPowerOn**( **newval**) |
| nodejs | function **set_enabledAtPowerOn**( **newval**) |
| php | function **set_enabledAtPowerOn**( **$newval**) |
| cpp | int **set_enabledAtPowerOn**( Y_ENABLEDATPOWERON_enum **newval**) |
| m | -(int) setEnabledAtPowerOn : (Y_ENABLEDATPOWERON_enum) **newval** |
| pas | function **set_enabledAtPowerOn**( **newval**: Integer): integer |
| vb | function **set_enabledAtPowerOn**( ByVal **newval** As Integer) As Integer |
| cs | int **set_enabledAtPowerOn**( int **newval**) |
| java | int **set_enabledAtPowerOn**( int **newval**) |
| py | def **set_enabledAtPowerOn**( **newval**) |
| cmd | YPwmOutput **target** **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.

## pwmoutput→set_frequency() pwmoutput→setFrequency()[pwmoutput setFrequency: ] YPwmOutput

Configures the PWM frequency.

| | |
|---|---|
| js | function **set_frequency**( **newval**) |
| nodejs | function **set_frequency**( **newval**) |
| php | function **set_frequency**( **$newval**) |
| cpp | int **set_frequency**( int **newval**) |
| m | -(int) setFrequency : (int) **newval** |
| pas | function **set_frequency**( **newval**: LongInt): integer |
| vb | function **set_frequency**( ByVal **newval** As Integer) As Integer |
| cs | int **set_frequency**( int **newval**) |
| java | int **set_frequency**( int **newval**) |
| py | def **set_frequency**( **newval**) |
| cmd | YPwmOutput **target** **set_frequency** **newval** |

The duty cycle is kept unchanged thanks to an automatic pulse width change.

**Parameters :**

**newval** an integer

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmoutput→set_logicalName() pwmoutput→setLogicalName()[pwmoutput setLogicalName: ] YPwmOutput

Changes the logical name of the PWM.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YPwmOutput **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the PWM.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## pwmoutput→set_period() pwmoutput→setPeriod()[pwmoutput setPeriod: ] YPwmOutput

Configures the PWM period.

| | |
|---|---|
| js | function **set_period**( **newval**) |
| nodejs | function **set_period**( **newval**) |
| php | function **set_period**( **$newval**) |
| cpp | int **set_period**( double **newval**) |
| m | -(int) setPeriod : (double) **newval** |
| pas | function **set_period**( **newval**: double): integer |
| vb | function **set_period**( ByVal **newval** As Double) As Integer |
| cs | int **set_period**( double **newval**) |
| java | int **set_period**( double **newval**) |
| py | def **set_period**( **newval**) |
| cmd | YPwmOutput **target** **set_period** **newval** |

**Parameters :**

**newval** a floating point number

**Returns :**

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmoutput→set_pulseDuration() pwmoutput→setPulseDuration()[pwmoutput setPulseDuration: ] YPwmOutput

Configures the PWM pluses length.

| | |
|---|---|
| js | function **set_pulseDuration**( **newval**) |
| nodejs | function **set_pulseDuration**( **newval**) |
| php | function **set_pulseDuration**( **$newval**) |
| cpp | int **set_pulseDuration**( double **newval**) |
| m | -(int) setPulseDuration : (double) **newval** |
| pas | function **set_pulseDuration**( **newval**: double): integer |
| vb | function **set_pulseDuration**( ByVal **newval** As Double) As Integer |
| cs | int **set_pulseDuration**( double **newval**) |
| java | int **set_pulseDuration**( double **newval**) |
| py | def **set_pulseDuration**( **newval**) |
| cmd | YPwmOutput **target** **set_pulseDuration** **newval** |

A pulse length cannot be longer than period, otherwise it is truncated.

**Parameters :**

**newval** a floating point number

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmoutput→set_userData() pwmoutput→setUserData()[pwmoutput setUserData: ] YPwmOutput

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## pwmoutput→wait_async() YPwmOutput

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

`js` function **wait_async**( **callback**, **context**)
`nodejs` function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.31. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_pwmpowersource.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YPwmPowerSource = yoctolib.YPwmPowerSource; |
| php | require_once('yocto_pwmpowersource.php'); |
| cpp | #include "yocto_pwmpowersource.h" |
| m | #import "yocto_pwmpowersource.h" |
| pas | uses yocto_pwmpowersource; |
| vb | yocto_pwmpowersource.vb |
| cs | yocto_pwmpowersource.cs |
| java | import com.yoctopuce.YoctoAPI.YPwmPowerSource; |
| py | from yocto_pwmpowersource import * |

**Global functions**

**yFindPwmPowerSource(func)**

Retrieves a voltage source for a given identifier.

**yFirstPwmPowerSource()**

Starts the enumeration of Voltage sources currently accessible.

**YPwmPowerSource methods**

**pwmpowersource→describe()**

Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**pwmpowersource→get_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

**pwmpowersource→get_errorMessage()**

Returns the error message of the latest error with the voltage source.

**pwmpowersource→get_errorType()**

Returns the numerical error code of the latest error with the voltage source.

**pwmpowersource→get_friendlyName()**

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

**pwmpowersource→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**pwmpowersource→get_functionId()**

Returns the hardware identifier of the voltage source, without reference to the module.

**pwmpowersource→get_hardwareId()**

Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

**pwmpowersource→get_logicalName()**

Returns the logical name of the voltage source.

**pwmpowersource→get_module()**

Gets the `YModule` object for the device on which the function is located.

**pwmpowersource→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**pwmpowersource→get_powerMode()**
Returns the selected power source for the PWM on the same device

**pwmpowersource→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**pwmpowersource→isOnline()**
Checks if the voltage source is currently reachable, without raising any error.

**pwmpowersource→isOnline_async(callback, context)**
Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

**pwmpowersource→load(msValidity)**
Preloads the voltage source cache with a specified validity duration.

**pwmpowersource→load_async(msValidity, callback, context)**
Preloads the voltage source cache with a specified validity duration (asynchronous version).

**pwmpowersource→nextPwmPowerSource()**
Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

**pwmpowersource→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**pwmpowersource→set_logicalName(newval)**
Changes the logical name of the voltage source.

**pwmpowersource→set_powerMode(newval)**
Changes the PWM power source.

**pwmpowersource→set_userData(data)**
Stores a user context provided as argument in the userData attribute of the function.

**pwmpowersource→wait_async(callback, context)**
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YPwmPowerSource.FindPwmPowerSource() yFindPwmPowerSource()yFindPwmPowerSource() YPwmPowerSource

Retrieves a voltage source for a given identifier.

| | |
|---|---|
| js | function **yFindPwmPowerSource**( **func**) |
| nodejs | function **FindPwmPowerSource**( **func**) |
| php | function **yFindPwmPowerSource**( **$func**) |
| cpp | YPwmPowerSource* **yFindPwmPowerSource**( const string& **func**) |
| m | YPwmPowerSource* **yFindPwmPowerSource**( NSString* **func**) |
| pas | function **yFindPwmPowerSource**( **func**: string): TYPwmPowerSource |
| vb | function **yFindPwmPowerSource**( ByVal **func** As String) As YPwmPowerSource |
| cs | YPwmPowerSource **FindPwmPowerSource**( string **func**) |
| java | YPwmPowerSource **FindPwmPowerSource**( String **func**) |
| py | def **FindPwmPowerSource**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmPowerSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the voltage source

**Returns :**

a `YPwmPowerSource` object allowing you to drive the voltage source.

## YPwmPowerSource.FirstPwmPowerSource() yFirstPwmPowerSource()yFirstPwmPowerSource() YPwmPowerSource

Starts the enumeration of Voltage sources currently accessible.

| | |
|---|---|
| js | function **yFirstPwmPowerSource**( ) |
| nodejs | function **FirstPwmPowerSource**( ) |
| php | function **yFirstPwmPowerSource**( ) |
| cpp | YPwmPowerSource* **yFirstPwmPowerSource**( ) |
| m | YPwmPowerSource* **yFirstPwmPowerSource**( ) |
| pas | function **yFirstPwmPowerSource**( ): TYPwmPowerSource |
| vb | function **yFirstPwmPowerSource**( ) As YPwmPowerSource |
| cs | YPwmPowerSource **FirstPwmPowerSource**( ) |
| java | YPwmPowerSource **FirstPwmPowerSource**( ) |
| py | def **FirstPwmPowerSource**( ) |

Use the method `YPwmPowerSource.nextPwmPowerSource()` to iterate on next Voltage sources.

**Returns :**

a pointer to a `YPwmPowerSource` object, corresponding to the first source currently online, or a `null` pointer if there are none.

### pwmpowersource→describe()[pwmpowersource describe] YPwmPowerSource

Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the voltage source (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## pwmpowersource→get_advertisedValue() pwmpowersource→advertisedValue() [pwmpowersource advertisedValue] YPwmPowerSource

Returns the current value of the voltage source (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YPwmPowerSource **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the voltage source (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## pwmpowersource→get_errorMessage() pwmpowersource→errorMessage() [pwmpowersource errorMessage]

**YPwmPowerSource**

Returns the error message of the latest error with the voltage source.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the voltage source object

## pwmpowersource→get_errorType() pwmpowersource→errorType() YPwmPowerSource

Returns the numerical error code of the latest error with the voltage source.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the voltage source object

## pwmpowersource→get_friendlyName() pwmpowersource→friendlyName()[pwmpowersource friendlyName] YPwmPowerSource

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the voltage source if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage source (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the voltage source using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## pwmpowersource→get_functionDescriptor() pwmpowersource→functionDescriptor() [pwmpowersource functionDescriptor] YPwmPowerSource

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## pwmpowersource→get_functionId() pwmpowersource→functionId()[pwmpowersource functionId] YPwmPowerSource

Returns the hardware identifier of the voltage source, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the voltage source (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## pwmpowersource→get_hardwareId() pwmpowersource→hardwareId()[pwmpowersource hardwareId] YPwmPowerSource

Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage source. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the voltage source (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## pwmpowersource→get_logicalName() pwmpowersource→logicalName()[pwmpowersource logicalName] YPwmPowerSource

Returns the logical name of the voltage source.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YPwmPowerSource **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the voltage source. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## pwmpowersource→get_module() pwmpowersource→module()[pwmpowersource module] YPwmPowerSource

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## pwmpowersource→get_module_async()
## pwmpowersource→module_async()

**YPwmPowerSource**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pwmpowersource→get_powerMode() pwmpowersource→powerMode()[pwmpowersource powerMode]

YPwmPowerSource

Returns the selected power source for the PWM on the same device

| | |
|---|---|
| js | function **get_powerMode**( ) |
| nodejs | function **get_powerMode**( ) |
| php | function **get_powerMode**( ) |
| cpp | Y_POWERMODE_enum **get_powerMode**( ) |
| m | -(Y_POWERMODE_enum) powerMode |
| pas | function **get_powerMode**( ): Integer |
| vb | function **get_powerMode**( ) As Integer |
| cs | int **get_powerMode**( ) |
| java | int **get_powerMode**( ) |
| py | def **get_powerMode**( ) |

**Returns :**

a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns `Y_POWERMODE_INVALID`.

## pwmpowersource→get_userData() pwmpowersource→userData()[pwmpowersource userData] YPwmPowerSource

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## pwmpowersource→isOnline()[pwmpowersource isOnline] YPwmPowerSource

Checks if the voltage source is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

**Returns :**

`true` if the voltage source can be reached, and `false` otherwise

## pwmpowersource→isOnline_async() YPwmPowerSource

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pwmpowersource→load()[pwmpowersource load: ] YPwmPowerSource

Preloads the voltage source cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## pwmpowersource→load_async() YPwmPowerSource

Preloads the voltage source cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## pwmpowersource→nextPwmPowerSource() [pwmpowersource nextPwmPowerSource] YPwmPowerSource

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

| | |
|---|---|
| js | function **nextPwmPowerSource**( ) |
| nodejs | function **nextPwmPowerSource**( ) |
| php | function **nextPwmPowerSource**( ) |
| cpp | YPwmPowerSource * **nextPwmPowerSource**( ) |
| m | -(YPwmPowerSource*) **nextPwmPowerSource** |
| pas | function **nextPwmPowerSource**( ): TYPwmPowerSource |
| vb | function **nextPwmPowerSource**( ) As YPwmPowerSource |
| cs | YPwmPowerSource **nextPwmPowerSource**( ) |
| java | YPwmPowerSource **nextPwmPowerSource**( ) |
| py | def **nextPwmPowerSource**( ) |

**Returns :**

a pointer to a `YPwmPowerSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more Voltage sources to enumerate.

## pwmpowersource→registerValueCallback() [pwmpowersource registerValueCallback: ]

**YPwmPowerSource**

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YPwmPowerSourceValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YPwmPowerSourceValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYPwmPowerSourceValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## pwmpowersource→set_logicalName() pwmpowersource→setLogicalName() [pwmpowersource setLogicalName: ]

**YPwmPowerSource**

Changes the logical name of the voltage source.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YPwmPowerSource **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the voltage source.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## pwmpowersource→set_powerMode() pwmpowersource→setPowerMode() [pwmpowersource setPowerMode: ]

**YPwmPowerSource**

Changes the PWM power source.

| | |
|---|---|
| js | function **set_powerMode**( **newval**) |
| nodejs | function **set_powerMode**( **newval**) |
| php | function **set_powerMode**( **$newval**) |
| cpp | int **set_powerMode**( Y_POWERMODE_enum **newval**) |
| m | -(int) setPowerMode : (Y_POWERMODE_enum) **newval** |
| pas | function **set_powerMode**( **newval**: Integer): integer |
| vb | function **set_powerMode**( ByVal **newval** As Integer) As Integer |
| cs | int **set_powerMode**( int **newval**) |
| java | int **set_powerMode**( int **newval**) |
| py | def **set_powerMode**( **newval**) |
| cmd | YPwmPowerSource **target set_powerMode newval** |

PWM can use isolated 5V from USB, isolated 3V from USb or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

**Parameters :**

**newval** a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the PWM power source

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## pwmpowersource→set_userData() pwmpowersource→setUserData()[pwmpowersource setUserData: ] YPwmPowerSource

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## pwmpowersource→wait_async() YPwmPowerSource

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.32. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_gyro.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YGyro = yoctolib.YGyro; |
| php | require_once('yocto_gyro.php'); |
| cpp | #include "yocto_gyro.h" |
| m | #import "yocto_gyro.h" |
| pas | uses yocto_gyro; |
| vb | yocto_gyro.vb |
| cs | yocto_gyro.cs |
| java | import com.yoctopuce.YoctoAPI.YGyro; |
| py | from yocto_gyro import * |

**Global functions**

**yFindQt(func)**

Retrieves a quaternion component for a given identifier.

**yFirstQt()**

Starts the enumeration of quaternion components currently accessible.

**YQt methods**

**qt→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**qt→describe()**

Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**qt→get_advertisedValue()**

Returns the current value of the quaternion component (no more than 6 characters).

**qt→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**qt→get_currentValue()**

Returns the current value of the value.

**qt→get_errorMessage()**

Returns the error message of the latest error with the quaternion component.

**qt→get_errorType()**

Returns the numerical error code of the latest error with the quaternion component.

**qt→get_friendlyName()**

Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.

**qt→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**qt→get_functionId()**

Returns the hardware identifier of the quaternion component, without reference to the module.

**qt→get_hardwareId()**

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

**qt→get_highestValue()**

Returns the maximal value observed for the value since the device was started.

**qt→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**qt→get_logicalName()**

Returns the logical name of the quaternion component.

**qt→get_lowestValue()**

Returns the minimal value observed for the value since the device was started.

**qt→get_module()**

Gets the `YModule` object for the device on which the function is located.

**qt→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**qt→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**qt→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**qt→get_resolution()**

Returns the resolution of the measured values.

**qt→get_unit()**

Returns the measuring unit for the value.

**qt→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**qt→isOnline()**

Checks if the quaternion component is currently reachable, without raising any error.

**qt→isOnline_async(callback, context)**

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

**qt→load(msValidity)**

Preloads the quaternion component cache with a specified validity duration.

**qt→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**qt→load_async(msValidity, callback, context)**

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

**qt→nextQt()**

Continues the enumeration of quaternion components started using `yFirstQt()`.

**qt→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**qt→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**qt→set_highestValue(newval)**

Changes the recorded maximal value observed.

**qt→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**qt→set_logicalName(newval)**

Changes the logical name of the quaternion component.

**qt→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**qt→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**qt→set_resolution(newval)**

Changes the resolution of the measured physical values.

**qt→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**qt→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YQt.FindQt() yFindQt()yFindQt()

YQt

Retrieves a quaternion component for a given identifier.

| | |
|---|---|
| js | function **yFindQt**( **func**) |
| nodejs | function **FindQt**( **func**) |
| php | function **yFindQt**( **$func**) |
| cpp | YQt* **yFindQt**( string **func**) |
| m | +(YQt*) **yFindQt** : (NSString*) **func** |
| pas | function **yFindQt**( **func**: string): TYQt |
| vb | function **yFindQt**( ByVal **func** As String) As YQt |
| cs | YQt **FindQt**( string **func**) |
| java | YQt **FindQt**( String **func**) |
| py | def **FindQt**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQt.isOnline()` to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the quaternion component

**Returns :**

a `YQt` object allowing you to drive the quaternion component.

## YQt.FirstQt() yFirstQt()yFirstQt() YQt

Starts the enumeration of quaternion components currently accessible.

| | |
|---|---|
| js | function **yFirstQt**( ) |
| nodejs | function **FirstQt**( ) |
| php | function **yFirstQt**( ) |
| cpp | YQt* **yFirstQt**( ) |
| m | YQt* **yFirstQt**( ) |
| pas | function **yFirstQt**( ): TYQt |
| vb | function **yFirstQt**( ) As YQt |
| cs | YQt **FirstQt**( ) |
| java | YQt **FirstQt**( ) |
| py | def **FirstQt**( ) |

Use the method `YQt.nextQt()` to iterate on next quaternion components.

**Returns :**

a pointer to a `YQt` object, corresponding to the first quaternion component currently online, or a `null` pointer if there are none.

## qt→calibrateFromPoints()[qt calibrateFromPoints: ] YQt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YSensor **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## qt→describe()[qt describe] YQt

Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the quaternion component (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## qt→get_advertisedValue() qt→advertisedValue()[qt advertisedValue] YQt

Returns the current value of the quaternion component (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YSensor **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the quaternion component (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## qt→get_currentRawValue() qt→currentRawValue()[qt currentRawValue] YQt

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YSensor **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## qt→get_currentValue() qt→currentValue()[qt currentValue] YQt

Returns the current value of the value.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YSensor **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current value of the value

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## qt→get_errorMessage()
## qt→errorMessage()[qt errorMessage] YQt

Returns the error message of the latest error with the quaternion component.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the quaternion component object

## qt→get_errorType() qt→errorType() YQt

Returns the numerical error code of the latest error with the quaternion component.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the quaternion component object

## qt→get_friendlyName() YQt
## qt→friendlyName()[qt friendlyName]

Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the quaternion component if they are defined, otherwise the serial number of the module and the hardware identifier of the quaternion component (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the quaternion component using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## qt→get_functionDescriptor() qt→functionDescriptor()[qt functionDescriptor] YQt

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## qt→get_functionId() qt→functionId()[qt functionId] YQt

Returns the hardware identifier of the quaternion component, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the quaternion component (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## qt→get_hardwareId() qt→hardwareId()[qt hardwareId] YQt

Returns the unique hardware identifier of the quaternion component in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quaternion component. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the quaternion component (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## qt→get_highestValue() qt→highestValue()[qt highestValue] YQt

Returns the maximal value observed for the value since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YSensor **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## qt→get_logFrequency() qt→logFrequency()[qt logFrequency] YQt

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YSensor **target** **get_logFrequency** |

**Returns :**
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## qt→get_logicalName() qt→logicalName()[qt logicalName] YQt

Returns the logical name of the quaternion component.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YSensor **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the quaternion component. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## qt→get_lowestValue() qt→lowestValue()[qt lowestValue] YQt

Returns the minimal value observed for the value since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YSensor **target** **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 module]

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## qt→get_module_async() qt→module_async() YQt

Gets the YModule object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## qt→get_recordedData() qt→recordedData()[qt recordedData: ] YQt

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YSensor **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## qt→get_reportFrequency() qt→reportFrequency()[qt reportFrequency] YQt

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YSensor **target** **get_reportFrequency** |

**Returns :**
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## qt→get_resolution() qt→resolution()[qt resolution] YQt

Returns the resolution of the measured values.

```
js      function get_resolution( )
nodejs  function get_resolution( )
php     function get_resolution( )
cpp     double get_resolution( )
m       -(double) resolution
pas     function get_resolution( ): double
vb      function get_resolution( ) As Double
cs      double get_resolution( )
java    double get_resolution( )
py      def get_resolution( )
cmd     YSensor target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## qt→get_unit() YQt
## qt→unit()[qt unit]

Returns the measuring unit for the value.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YSensor **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the value

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## qt→get_userData() qt→userData()[qt userData] YQt

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## qt→isOnline()[qt isOnline] YQt

Checks if the quaternion component is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

**Returns :**
`true` if the quaternion component can be reached, and `false` otherwise

## qt→isOnline_async() YQt

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## qt→load()[qt load: ] YQt

Preloads the quaternion component cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## qt→loadCalibrationPoints()[qt loadCalibrationPoints: ] YQt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YSensor **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## qt→load_async() YQt

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## qt→nextQt()[qt nextQt] YQt

Continues the enumeration of quaternion components started using `yFirstQt()`.

| | |
|---|---|
| js | function **nextQt**( ) |
| nodejs | function **nextQt**( ) |
| php | function **nextQt**( ) |
| cpp | YQt * **nextQt**( ) |
| m | -(YQt*) **nextQt** |
| pas | function **nextQt**( ): TYQt |
| vb | function **nextQt**( ) As YQt |
| cs | YQt **nextQt**( ) |
| java | YQt **nextQt**( ) |
| py | def **nextQt**( ) |

**Returns :**

a pointer to a `YQt` object, corresponding to a quaternion component currently online, or a `null` pointer if there are no more quaternion components to enumerate.

## qt→registerTimedReportCallback()[qt registerTimedReportCallback: ] YQt

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YQtTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YQtTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYQtTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## qt→registerValueCallback()[qt registerValueCallback: ] YQt

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YQtValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YQtValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYQtValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## qt→set_highestValue() qt→setHighestValue()[qt setHighestValue: ] YQt

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YSensor **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## qt→set_logFrequency() qt→setLogFrequency()[qt setLogFrequency: ] YQt

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YSensor **target set_logFrequency newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## qt→set_logicalName() qt→setLogicalName()[qt setLogicalName: ] YQt

Changes the logical name of the quaternion component.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YSensor **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the quaternion component.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## qt→set_lowestValue() qt→setLowestValue()[qt setLowestValue: ] YQt

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YSensor **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

### qt→set_reportFrequency() qt→setReportFrequency()[qt setReportFrequency: ] YQt

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YSensor **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## qt→set_resolution() qt→setResolution()[qt setResolution: ] YQt

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YSensor **target set_resolution newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## qt→set_userData() qt→setUserData()[qt setUserData: ] YQt

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## qt→wait_async() YQt

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.33. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_realtimeclock.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YRealTimeClock = yoctolib.YRealTimeClock; |
| php | require_once('yocto_realtimeclock.php'); |
| cpp | #include "yocto_realtimeclock.h" |
| m | #import "yocto_realtimeclock.h" |
| pas | uses yocto_realtimeclock; |
| vb | yocto_realtimeclock.vb |
| cs | yocto_realtimeclock.cs |
| java | import com.yoctopuce.YoctoAPI.YRealTimeClock; |
| py | from yocto_realtimeclock import * |

| Global functions |
|---|
| **yFindRealTimeClock(func)**<br>Retrieves a clock for a given identifier. |
| **yFirstRealTimeClock()**<br>Starts the enumeration of clocks currently accessible. |
| **`YRealTimeClock` methods** |
| **realtimeclock→describe()**<br>Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **realtimeclock→get_advertisedValue()**<br>Returns the current value of the clock (no more than 6 characters). |
| **realtimeclock→get_dateTime()**<br>Returns the current time in the form "YYYY/MM/DD hh:mm:ss" |
| **realtimeclock→get_errorMessage()**<br>Returns the error message of the latest error with the clock. |
| **realtimeclock→get_errorType()**<br>Returns the numerical error code of the latest error with the clock. |
| **realtimeclock→get_friendlyName()**<br>Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`. |
| **realtimeclock→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **realtimeclock→get_functionId()**<br>Returns the hardware identifier of the clock, without reference to the module. |
| **realtimeclock→get_hardwareId()**<br>Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`. |
| **realtimeclock→get_logicalName()**<br>Returns the logical name of the clock. |
| **realtimeclock→get_module()** |

Gets the `YModule` object for the device on which the function is located.

**realtimeclock→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**realtimeclock→get_timeSet()**

Returns true if the clock has been set, and false otherwise.

**realtimeclock→get_unixTime()**

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

**realtimeclock→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**realtimeclock→get_utcOffset()**

Returns the number of seconds between current time and UTC time (time zone).

**realtimeclock→isOnline()**

Checks if the clock is currently reachable, without raising any error.

**realtimeclock→isOnline_async(callback, context)**

Checks if the clock is currently reachable, without raising any error (asynchronous version).

**realtimeclock→load(msValidity)**

Preloads the clock cache with a specified validity duration.

**realtimeclock→load_async(msValidity, callback, context)**

Preloads the clock cache with a specified validity duration (asynchronous version).

**realtimeclock→nextRealTimeClock()**

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

**realtimeclock→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**realtimeclock→set_logicalName(newval)**

Changes the logical name of the clock.

**realtimeclock→set_unixTime(newval)**

Changes the current time.

**realtimeclock→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**realtimeclock→set_utcOffset(newval)**

Changes the number of seconds between current time and UTC time (time zone).

**realtimeclock→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YRealTimeClock.FindRealTimeClock() yFindRealTimeClock()yFindRealTimeClock() YRealTimeClock

Retrieves a clock for a given identifier.

| | |
|---|---|
| js | function **yFindRealTimeClock**( **func**) |
| nodejs | function **FindRealTimeClock**( **func**) |
| php | function **yFindRealTimeClock**( **$func**) |
| cpp | YRealTimeClock* **yFindRealTimeClock**( const string& **func**) |
| m | YRealTimeClock* **yFindRealTimeClock**( NSString* **func**) |
| pas | function **yFindRealTimeClock**( **func**: string): TYRealTimeClock |
| vb | function **yFindRealTimeClock**( ByVal **func** As String) As YRealTimeClock |
| cs | YRealTimeClock **FindRealTimeClock**( string **func**) |
| java | YRealTimeClock **FindRealTimeClock**( String **func**) |
| py | def **FindRealTimeClock**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the clock is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRealTimeClock.isOnline()` to test if the clock is indeed online at a given time. In case of ambiguity when looking for a clock by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the clock

**Returns :**

a `YRealTimeClock` object allowing you to drive the clock.

## YRealTimeClock.FirstRealTimeClock() yFirstRealTimeClock()yFirstRealTimeClock() YRealTimeClock

Starts the enumeration of clocks currently accessible.

| | |
|---|---|
| js | function **yFirstRealTimeClock**( ) |
| nodejs | function **FirstRealTimeClock**( ) |
| php | function **yFirstRealTimeClock**( ) |
| cpp | YRealTimeClock* **yFirstRealTimeClock**( ) |
| m | YRealTimeClock* **yFirstRealTimeClock**( ) |
| pas | function **yFirstRealTimeClock**( ): TYRealTimeClock |
| vb | function **yFirstRealTimeClock**( ) As YRealTimeClock |
| cs | YRealTimeClock **FirstRealTimeClock**( ) |
| java | YRealTimeClock **FirstRealTimeClock**( ) |
| py | def **FirstRealTimeClock**( ) |

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

**Returns :**

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a `null` pointer if there are none.

## realtimeclock→describe()[realtimeclock describe] YRealTimeClock

Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the clock (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## realtimeclock→get_advertisedValue() realtimeclock→advertisedValue()[realtimeclock advertisedValue] YRealTimeClock

Returns the current value of the clock (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YRealTimeClock **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the clock (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## realtimeclock→get_dateTime() realtimeclock→dateTime()[realtimeclock dateTime] YRealTimeClock

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

| | |
|---|---|
| js | function **get_dateTime**( ) |
| nodejs | function **get_dateTime**( ) |
| php | function **get_dateTime**( ) |
| cpp | string **get_dateTime**( ) |
| m | -(NSString*) dateTime |
| pas | function **get_dateTime**( ): string |
| vb | function **get_dateTime**( ) As String |
| cs | string **get_dateTime**( ) |
| java | String **get_dateTime**( ) |
| py | def **get_dateTime**( ) |

**Returns :**
a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns `Y_DATETIME_INVALID`.

## realtimeclock→get_errorMessage() realtimeclock→errorMessage()[realtimeclock errorMessage] YRealTimeClock

Returns the error message of the latest error with the clock.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the clock object

## realtimeclock→get_errorType() realtimeclock→errorType() YRealTimeClock

Returns the numerical error code of the latest error with the clock.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the clock object

## realtimeclock→get_friendlyName() realtimeclock→friendlyName()[realtimeclock friendlyName] YRealTimeClock

Returns a global identifier of the clock in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the clock if they are defined, otherwise the serial number of the module and the hardware identifier of the clock (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the clock using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## realtimeclock→get_functionDescriptor() realtimeclock→functionDescriptor()[realtimeclock functionDescriptor] YRealTimeClock

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## realtimeclock→get_functionId() realtimeclock→functionId()[realtimeclock functionId] YRealTimeClock

Returns the hardware identifier of the clock, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the clock (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## realtimeclock→get_hardwareId() realtimeclock→hardwareId()[realtimeclock hardwareId] YRealTimeClock

Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the clock. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the clock (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## realtimeclock→get_logicalName() realtimeclock→logicalName()[realtimeclock logicalName] YRealTimeClock

Returns the logical name of the clock.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YRealTimeClock **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the clock. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## realtimeclock→get_module() realtimeclock→module()[realtimeclock module] YRealTimeClock

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## realtimeclock→get_module_async() realtimeclock→module_async() YRealTimeClock

Gets the `YModule` object for the device on which the function is located (asynchronous version).

js function **get_module_async**( **callback**, **context**)
nodejs function **get_module_async**( **callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## realtimeclock→get_timeSet() realtimeclock→timeSet()[realtimeclock timeSet] YRealTimeClock

Returns true if the clock has been set, and false otherwise.

| | |
|---|---|
| js | function **get_timeSet**( ) |
| nodejs | function **get_timeSet**( ) |
| php | function **get_timeSet**( ) |
| cpp | Y_TIMESET_enum **get_timeSet**( ) |
| m | -(Y_TIMESET_enum) timeSet |
| pas | function **get_timeSet**( ): Integer |
| vb | function **get_timeSet**( ) As Integer |
| cs | int **get_timeSet**( ) |
| java | int **get_timeSet**( ) |
| py | def **get_timeSet**( ) |
| cmd | YRealTimeClock **target** **get_timeSet** |

**Returns :**

either `Y_TIMESET_FALSE` or `Y_TIMESET_TRUE`, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns `Y_TIMESET_INVALID`.

## realtimeclock→get_unixTime() realtimeclock→unixTime()[realtimeclock unixTime] YRealTimeClock

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

| | |
|---|---|
| js | function **get_unixTime**( ) |
| nodejs | function **get_unixTime**( ) |
| php | function **get_unixTime**( ) |
| cpp | s64 **get_unixTime**( ) |
| m | -(s64) unixTime |
| pas | function **get_unixTime**( ): int64 |
| vb | function **get_unixTime**( ) As Long |
| cs | long **get_unixTime**( ) |
| java | long **get_unixTime**( ) |
| py | def **get_unixTime**( ) |
| cmd | YRealTimeClock **target** **get_unixTime** |

**Returns :**

an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns `Y_UNIXTIME_INVALID`.

## realtimeclock→get_userData() realtimeclock→userData()[realtimeclock userData] YRealTimeClock

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## realtimeclock→get_utcOffset() realtimeclock→utcOffset()[realtimeclock utcOffset] YRealTimeClock

Returns the number of seconds between current time and UTC time (time zone).

| | |
|---|---|
| js | function **get_utcOffset**( ) |
| nodejs | function **get_utcOffset**( ) |
| php | function **get_utcOffset**( ) |
| cpp | int **get_utcOffset**( ) |
| m | -(int) utcOffset |
| pas | function **get_utcOffset**( ): LongInt |
| vb | function **get_utcOffset**( ) As Integer |
| cs | int **get_utcOffset**( ) |
| java | int **get_utcOffset**( ) |
| py | def **get_utcOffset**( ) |
| cmd | YRealTimeClock **target** **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.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

**Returns :**

`true` if the clock can be reached, and `false` otherwise

## realtimeclock→isOnline_async() YRealTimeClock

Checks if the clock is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## realtimeclock→load()[realtimeclock load: ] YRealTimeClock

Preloads the clock cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## realtimeclock→load_async() YRealTimeClock

Preloads the clock cache with a specified validity duration (asynchronous version).

`js` function **load_async**( **msValidity**, **callback**, **context**)
`nodejs` function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## realtimeclock→nextRealTimeClock()[realtimeclock nextRealTimeClock] YRealTimeClock

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

| | |
|---|---|
| js | function **nextRealTimeClock**( ) |
| nodejs | function **nextRealTimeClock**( ) |
| php | function **nextRealTimeClock**( ) |
| cpp | YRealTimeClock * **nextRealTimeClock**( ) |
| m | -(YRealTimeClock*) **nextRealTimeClock** |
| pas | function **nextRealTimeClock**( ): TYRealTimeClock |
| vb | function **nextRealTimeClock**( ) As YRealTimeClock |
| cs | YRealTimeClock **nextRealTimeClock**( ) |
| java | YRealTimeClock **nextRealTimeClock**( ) |
| py | def **nextRealTimeClock**( ) |

**Returns :**

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a `null` pointer if there are no more clocks to enumerate.

## realtimeclock→registerValueCallback()[realtimeclock registerValueCallback: ] YRealTimeClock

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YRealTimeClockValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YRealTimeClockValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYRealTimeClockValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## realtimeclock→set_logicalName() realtimeclock→setLogicalName()[realtimeclock setLogicalName: ] YRealTimeClock

Changes the logical name of the clock.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YRealTimeClock **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the clock.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## realtimeclock→set_unixTime() realtimeclock→setUnixTime()[realtimeclock setUnixTime: ] YRealTimeClock

Changes the current time.

| | |
|---|---|
| js | function **set_unixTime**( **newval**) |
| nodejs | function **set_unixTime**( **newval**) |
| php | function **set_unixTime**( **$newval**) |
| cpp | int **set_unixTime**( s64 **newval**) |
| m | -(int) setUnixTime : (s64) **newval** |
| pas | function **set_unixTime**( **newval**: int64): integer |
| vb | function **set_unixTime**( ByVal **newval** As Long) As Integer |
| cs | int **set_unixTime**( long **newval**) |
| java | int **set_unixTime**( long **newval**) |
| py | def **set_unixTime**( **newval**) |
| cmd | YRealTimeClock **target** **set_unixTime** **newval** |

Time is specifid in Unix format (number of elapsed seconds since Jan 1st, 1970). If current UTC time is known, utcOffset will be automatically adjusted for the new specified time.

**Parameters :**

**newval** an integer corresponding to the current time

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## realtimeclock→set_userData() realtimeclock→setUserData()[realtimeclock setUserData: ] YRealTimeClock

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## realtimeclock→set_utcOffset() realtimeclock→setUtcOffset()[realtimeclock setUtcOffset: ] YRealTimeClock

Changes the number of seconds between current time and UTC time (time zone).

| | |
|---|---|
| js | function **set_utcOffset**( **newval**) |
| nodejs | function **set_utcOffset**( **newval**) |
| php | function **set_utcOffset**( **$newval**) |
| cpp | int **set_utcOffset**( int **newval**) |
| m | -(int) setUtcOffset : (int) **newval** |
| pas | function **set_utcOffset**( **newval**: LongInt): integer |
| vb | function **set_utcOffset**( ByVal **newval** As Integer) As Integer |
| cs | int **set_utcOffset**( int **newval**) |
| java | int **set_utcOffset**( int **newval**) |
| py | def **set_utcOffset**( **newval**) |
| cmd | YRealTimeClock **target** **set_utcOffset** **newval** |

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time will automatically be updated according to the selected time zone.

**Parameters :**

**newval** an integer corresponding to the number of seconds between current time and UTC time (time zone)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## realtimeclock→wait_async() YRealTimeClock

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.34. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_refframe.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YRefFrame = yoctolib.YRefFrame; |
| php | require_once('yocto_refframe.php'); |
| cpp | #include "yocto_refframe.h" |
| m | #import "yocto_refframe.h" |
| pas | uses yocto_refframe; |
| vb | yocto_refframe.vb |
| cs | yocto_refframe.cs |
| java | import com.yoctopuce.YoctoAPI.YRefFrame; |
| py | from yocto_refframe import * |

**Global functions**

**yFindRefFrame(func)**

Retrieves a reference frame for a given identifier.

**yFirstRefFrame()**

Starts the enumeration of reference frames currently accessible.

**`YRefFrame` methods**

**refframe→cancel3DCalibration()**

Aborts the sensors tridimensional calibration process et restores normal settings.

**refframe→describe()**

Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**refframe→get_3DCalibrationHint()**

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

**refframe→get_3DCalibrationLogMsg()**

Returns the latest log message from the calibration process.

**refframe→get_3DCalibrationProgress()**

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

**refframe→get_3DCalibrationStage()**

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

**refframe→get_3DCalibrationStageProgress()**

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

**refframe→get_advertisedValue()**

Returns the current value of the reference frame (no more than 6 characters).

**refframe→get_bearing()**

Returns the reference bearing used by the compass.

**refframe→get_errorMessage()**

Returns the error message of the latest error with the reference frame.

**refframe→get_errorType()**

Returns the numerical error code of the latest error with the reference frame.

**refframe→get_friendlyName()**

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

**refframe→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**refframe→get_functionId()**

Returns the hardware identifier of the reference frame, without reference to the module.

**refframe→get_hardwareId()**

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

**refframe→get_logicalName()**

Returns the logical name of the reference frame.

**refframe→get_module()**

Gets the `YModule` object for the device on which the function is located.

**refframe→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**refframe→get_mountOrientation()**

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

**refframe→get_mountPosition()**

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

**refframe→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**refframe→isOnline()**

Checks if the reference frame is currently reachable, without raising any error.

**refframe→isOnline_async(callback, context)**

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

**refframe→load(msValidity)**

Preloads the reference frame cache with a specified validity duration.

**refframe→load_async(msValidity, callback, context)**

Preloads the reference frame cache with a specified validity duration (asynchronous version).

**refframe→more3DCalibration()**

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

**refframe→nextRefFrame()**

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

**refframe→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**refframe→save3DCalibration()**

Applies the sensors tridimensional calibration parameters that have just been computed.

**refframe→set_bearing(newval)**

Changes the reference bearing used by the compass.

**refframe→set_logicalName(newval)**

Changes the logical name of the reference frame.

**refframe→set_mountPosition(position, orientation)**

Changes the compass and tilt sensor frame of reference.

**refframe→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**refframe→start3DCalibration()**

Initiates the sensors tridimensional calibration process.

**refframe→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YRefFrame.FindRefFrame() yFindRefFrame()yFindRefFrame() YRefFrame

Retrieves a reference frame for a given identifier.

| | |
|---|---|
| js | function **yFindRefFrame**( **func**) |
| nodejs | function **FindRefFrame**( **func**) |
| php | function **yFindRefFrame**( **$func**) |
| cpp | YRefFrame* **yFindRefFrame**( const string& **func**) |
| m | YRefFrame* **yFindRefFrame**( NSString* **func**) |
| pas | function **yFindRefFrame**( **func**: string): TYRefFrame |
| vb | function **yFindRefFrame**( ByVal **func** As String) As YRefFrame |
| cs | YRefFrame **FindRefFrame**( string **func**) |
| java | YRefFrame **FindRefFrame**( String **func**) |
| py | def **FindRefFrame**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the reference frame

**Returns :**

a `YRefFrame` object allowing you to drive the reference frame.

## YRefFrame.FirstRefFrame() yFirstRefFrame()yFirstRefFrame() YRefFrame

Starts the enumeration of reference frames currently accessible.

| | |
|---|---|
| js | function **yFirstRefFrame**( ) |
| nodejs | function **FirstRefFrame**( ) |
| php | function **yFirstRefFrame**( ) |
| cpp | YRefFrame* **yFirstRefFrame**( ) |
| m | YRefFrame* **yFirstRefFrame**( ) |
| pas | function **yFirstRefFrame**( ): TYRefFrame |
| vb | function **yFirstRefFrame**( ) As YRefFrame |
| cs | YRefFrame **FirstRefFrame**( ) |
| java | YRefFrame **FirstRefFrame**( ) |
| py | def **FirstRefFrame**( ) |

Use the method `YRefFrame.nextRefFrame()` to iterate on next reference frames.

**Returns :**

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

## refframe→cancel3DCalibration()[refframe cancel3DCalibration] YRefFrame

Aborts the sensors tridimensional calibration process et restores normal settings.

| | |
|---|---|
| js | function **cancel3DCalibration**( ) |
| nodejs | function **cancel3DCalibration**( ) |
| php | function **cancel3DCalibration**( ) |
| cpp | int **cancel3DCalibration**( ) |
| m | -(int) **cancel3DCalibration** |
| pas | function **cancel3DCalibration**( ): LongInt |
| vb | function **cancel3DCalibration**( ) As Integer |
| cs | int **cancel3DCalibration**( ) |
| java | int **cancel3DCalibration**( ) |
| py | def **cancel3DCalibration**( ) |
| cmd | YRefFrame **target** **cancel3DCalibration** |

On failure, throws an exception or returns a negative error code.

## refframe→describe()[refframe describe] YRefFrame

Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE(NAME)=SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the reference frame (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## refframe→get_3DCalibrationHint() refframe→3DCalibrationHint()[refframe 3DCalibrationHint] YRefFrame

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

| | |
|---|---|
| `js` | function **get_3DCalibrationHint**( ) |
| `nodejs` | function **get_3DCalibrationHint**( ) |
| `php` | function **get_3DCalibrationHint**( ) |
| `cpp` | string **get_3DCalibrationHint**( ) |
| `m` | -(NSString*) 3DCalibrationHint |
| `pas` | function **get_3DCalibrationHint**( ): string |
| `vb` | function **get_3DCalibrationHint**( ) As String |
| `cs` | string **get_3DCalibrationHint**( ) |
| `java` | String **get_3DCalibrationHint**( ) |
| `py` | def **get_3DCalibrationHint**( ) |
| `cmd` | YRefFrame **target** **get_3DCalibrationHint** |

**Returns :**
a character string.

## refframe→get_3DCalibrationLogMsg() refframe→3DCalibrationLogMsg()[refframe 3DCalibrationLogMsg] YRefFrame

Returns the latest log message from the calibration process.

| | |
|---|---|
| js | function **get_3DCalibrationLogMsg**( ) |
| nodejs | function **get_3DCalibrationLogMsg**( ) |
| php | function **get_3DCalibrationLogMsg**( ) |
| cpp | string **get_3DCalibrationLogMsg**( ) |
| m | -(NSString*) 3DCalibrationLogMsg |
| pas | function **get_3DCalibrationLogMsg**( ): string |
| vb | function **get_3DCalibrationLogMsg**( ) As String |
| cs | string **get_3DCalibrationLogMsg**( ) |
| java | String **get_3DCalibrationLogMsg**( ) |
| py | def **get_3DCalibrationLogMsg**( ) |
| cmd | YRefFrame **target** **get_3DCalibrationLogMsg** |

When no new message is available, returns an empty string.

**Returns :**
a character string.

## refframe→get_3DCalibrationProgress() refframe→3DCalibrationProgress()[refframe 3DCalibrationProgress] YRefFrame

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

| | |
|---|---|
| js | function **get_3DCalibrationProgress**( ) |
| nodejs | function **get_3DCalibrationProgress**( ) |
| php | function **get_3DCalibrationProgress**( ) |
| cpp | int **get_3DCalibrationProgress**( ) |
| m | -(int) 3DCalibrationProgress |
| pas | function **get_3DCalibrationProgress**( ): LongInt |
| vb | function **get_3DCalibrationProgress**( ) As Integer |
| cs | int **get_3DCalibrationProgress**( ) |
| java | int **get_3DCalibrationProgress**( ) |
| py | def **get_3DCalibrationProgress**( ) |
| cmd | YRefFrame **target** **get_3DCalibrationProgress** |

**Returns :**
an integer between 0 (not started) and 100 (stage completed).

## refframe→get_3DCalibrationStage() refframe→3DCalibrationStage()[refframe 3DCalibrationStage] YRefFrame

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

| | |
|---|---|
| js | function **get_3DCalibrationStage**( ) |
| nodejs | function **get_3DCalibrationStage**( ) |
| php | function **get_3DCalibrationStage**( ) |
| cpp | int **get_3DCalibrationStage**( ) |
| m | -(int) 3DCalibrationStage |
| pas | function **get_3DCalibrationStage**( ): LongInt |
| vb | function **get_3DCalibrationStage**( ) As Integer |
| cs | int **get_3DCalibrationStage**( ) |
| java | int **get_3DCalibrationStage**( ) |
| py | def **get_3DCalibrationStage**( ) |
| cmd | YRefFrame **target** **get_3DCalibrationStage** |

**Returns :**
an integer, growing each time a calibration stage is completed.

## refframe→get_3DCalibrationStageProgress() refframe→3DCalibrationStageProgress()[refframe 3DCalibrationStageProgress] YRefFrame

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

| | |
|---|---|
| js | function **get_3DCalibrationStageProgress**( ) |
| nodejs | function **get_3DCalibrationStageProgress**( ) |
| php | function **get_3DCalibrationStageProgress**( ) |
| cpp | int **get_3DCalibrationStageProgress**( ) |
| m | -(int) 3DCalibrationStageProgress |
| pas | function **get_3DCalibrationStageProgress**( ): LongInt |
| vb | function **get_3DCalibrationStageProgress**( ) As Integer |
| cs | int **get_3DCalibrationStageProgress**( ) |
| java | int **get_3DCalibrationStageProgress**( ) |
| py | def **get_3DCalibrationStageProgress**( ) |
| cmd | YRefFrame **target** **get_3DCalibrationStageProgress** |

**Returns :**

an integer between 0 (not started) and 100 (stage completed).

## refframe→get_advertisedValue() refframe→advertisedValue()[refframe advertisedValue] YRefFrame

Returns the current value of the reference frame (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YRefFrame **target** **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 bearing]

Returns the reference bearing used by the compass.

| | |
|---|---|
| js | function **get_bearing**( ) |
| nodejs | function **get_bearing**( ) |
| php | function **get_bearing**( ) |
| cpp | double **get_bearing**( ) |
| m | -(double) bearing |
| pas | function **get_bearing**( ): double |
| vb | function **get_bearing**( ) As Double |
| cs | double **get_bearing**( ) |
| java | double **get_bearing**( ) |
| py | def **get_bearing**( ) |
| cmd | YRefFrame **target** **get_bearing** |

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

**Returns :**
a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns `Y_BEARING_INVALID`.

## refframe→get_errorMessage() YRefFrame
## refframe→errorMessage()[refframe errorMessage]

Returns the error message of the latest error with the reference frame.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the reference frame object

## refframe→get_errorType() refframe→errorType()

**YRefFrame**

Returns the numerical error code of the latest error with the reference frame.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the reference frame object

## refframe→get_friendlyName() refframe→friendlyName()[refframe friendlyName] YRefFrame

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the reference frame if they are defined, otherwise the serial number of the module and the hardware identifier of the reference frame (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the reference frame using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## refframe→get_functionDescriptor() refframe→functionDescriptor()[refframe functionDescriptor]

**YRefFrame**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## refframe→get_functionId() refframe→functionId()[refframe functionId] YRefFrame

Returns the hardware identifier of the reference frame, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the reference frame (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## refframe→get_hardwareId() refframe→hardwareId()[refframe hardwareId] YRefFrame

Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the reference frame (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## refframe→get_logicalName() refframe→logicalName()[refframe logicalName] YRefFrame

Returns the logical name of the reference frame.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YRefFrame **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the reference frame. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## refframe→get_module() refframe→module()[refframe module] YRefFrame

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## refframe→get_module_async() refframe→module_async() YRefFrame

Gets the YModule object for the device on which the function is located (asynchronous version).

js function **get_module_async**( **callback**, **context**)
nodejs function **get_module_async**( **callback**, **context**)

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## refframe→get_mountOrientation() refframe→mountOrientation()[refframe mountOrientation] **YRefFrame**

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

| | |
|---|---|
| js | function **get_mountOrientation**( ) |
| nodejs | function **get_mountOrientation**( ) |
| php | function **get_mountOrientation**( ) |
| cpp | Y_MOUNTORIENTATION **get_mountOrientation**( ) |
| m | -(Y_MOUNTORIENTATION) mountOrientation |
| pas | function **get_mountOrientation**( ): TYMOUNTORIENTATION |
| vb | function **get_mountOrientation**( ) As Y_MOUNTORIENTATION |
| cs | MOUNTORIENTATION **get_mountOrientation**( ) |
| java | MOUNTORIENTATION **get_mountOrientation**( ) |
| py | def **get_mountOrientation**( ) |
| cmd | YRefFrame **target** **get_mountOrientation** |

**Returns :**

a value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`, `Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

## refframe→get_mountPosition() refframe→mountPosition()[refframe mountPosition] YRefFrame

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

| | |
|---|---|
| js | function **get_mountPosition**( ) |
| nodejs | function **get_mountPosition**( ) |
| php | function **get_mountPosition**( ) |
| cpp | Y_MOUNTPOSITION **get_mountPosition**( ) |
| m | -(Y_MOUNTPOSITION) mountPosition |
| pas | function **get_mountPosition**( ): TYMOUNTPOSITION |
| vb | function **get_mountPosition**( ) As Y_MOUNTPOSITION |
| cs | MOUNTPOSITION **get_mountPosition**( ) |
| java | MOUNTPOSITION **get_mountPosition**( ) |
| py | def **get_mountPosition**( ) |
| cmd | YRefFrame **target** **get_mountPosition** |

**Returns :**

a value among the `Y_MOUNTPOSITION` enumeration (`Y_MOUNTPOSITION_BOTTOM`, `Y_MOUNTPOSITION_TOP`, `Y_MOUNTPOSITION_FRONT`, `Y_MOUNTPOSITION_RIGHT`, `Y_MOUNTPOSITION_REAR`, `Y_MOUNTPOSITION_LEFT`), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.

## refframe→get_userData() refframe→userData()[refframe userData] YRefFrame

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## refframe→isOnline()[refframe isOnline] YRefFrame

Checks if the reference frame is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

**Returns :**
`true` if the reference frame can be reached, and `false` otherwise

## refframe→isOnline_async() YRefFrame

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## refframe→load()[refframe load: ] YRefFrame

Preloads the reference frame cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## refframe→load_async() YRefFrame

Preloads the reference frame cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

### refframe→more3DCalibration()[refframe more3DCalibration] YRefFrame

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

| | |
|---|---|
| js | function **more3DCalibration**( ) |
| nodejs | function **more3DCalibration**( ) |
| php | function **more3DCalibration**( ) |
| cpp | int **more3DCalibration**( ) |
| m | -(int) **more3DCalibration** |
| pas | function **more3DCalibration**( ): LongInt |
| vb | function **more3DCalibration**( ) As Integer |
| cs | int **more3DCalibration**( ) |
| java | int **more3DCalibration**( ) |
| py | def **more3DCalibration**( ) |
| cmd | YRefFrame **target** **more3DCalibration** |

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.

## refframe→nextRefFrame()[refframe nextRefFrame] YRefFrame

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

| | |
|---|---|
| js | function **nextRefFrame**( ) |
| nodejs | function **nextRefFrame**( ) |
| php | function **nextRefFrame**( ) |
| cpp | YRefFrame * **nextRefFrame**( ) |
| m | -(YRefFrame*) **nextRefFrame** |
| pas | function **nextRefFrame**( ): TYRefFrame |
| vb | function **nextRefFrame**( ) As YRefFrame |
| cs | YRefFrame **nextRefFrame**( ) |
| java | YRefFrame **nextRefFrame**( ) |
| py | def **nextRefFrame**( ) |

**Returns :**

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a `null` pointer if there are no more reference frames to enumerate.

## refframe→registerValueCallback()[refframe registerValueCallback: ] YRefFrame

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YRefFrameValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YRefFrameValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYRefFrameValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## refframe→save3DCalibration()[refframe save3DCalibration] YRefFrame

Applies the sensors tridimensional calibration parameters that have just been computed.

| | |
|---|---|
| js | function **save3DCalibration**( ) |
| nodejs | function **save3DCalibration**( ) |
| php | function **save3DCalibration**( ) |
| cpp | int **save3DCalibration**( ) |
| m | -(int) **save3DCalibration** |
| pas | function **save3DCalibration**( ): LongInt |
| vb | function **save3DCalibration**( ) As Integer |
| cs | int **save3DCalibration**( ) |
| java | int **save3DCalibration**( ) |
| py | def **save3DCalibration**( ) |
| cmd | YRefFrame **target** **save3DCalibration** |

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

## refframe→set_bearing() refframe→setBearing()[refframe setBearing: ] YRefFrame

Changes the reference bearing used by the compass.

| | |
|---|---|
| js | function **set_bearing**( **newval**) |
| nodejs | function **set_bearing**( **newval**) |
| php | function **set_bearing**( **$newval**) |
| cpp | int **set_bearing**( double **newval**) |
| m | -(int) setBearing : (double) **newval** |
| pas | function **set_bearing**( **newval**: double): integer |
| vb | function **set_bearing**( ByVal **newval** As Double) As Integer |
| cs | int **set_bearing**( double **newval**) |
| java | int **set_bearing**( double **newval**) |
| py | def **set_bearing**( **newval**) |
| cmd | YRefFrame **target** **set_bearing** **newval** |

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a floating point number corresponding to the reference bearing used by the compass

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## refframe→set_logicalName() refframe→setLogicalName()[refframe setLogicalName: ] YRefFrame

Changes the logical name of the reference frame.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YRefFrame **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the reference frame.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## refframe→set_mountPosition() refframe→setMountPosition()[refframe setMountPosition: ] YRefFrame

Changes the compass and tilt sensor frame of reference.

| | |
|---|---|
| js | function **set_mountPosition**( **position**, **orientation**) |
| nodejs | function **set_mountPosition**( **position**, **orientation**) |
| php | function **set_mountPosition**( **$position**, **$orientation**) |
| cpp | int **set_mountPosition**( Y_MOUNTPOSITION **position**, Y_MOUNTORIENTATION **orientation**) |
| m | -(int) setMountPosition : (Y_MOUNTPOSITION) **position** : (Y_MOUNTORIENTATION) **orientation** |
| pas | function **set_mountPosition**( **position**: TYMOUNTPOSITION, **orientation**: TYMOUNTORIENTATION): LongInt |
| vb | function **set_mountPosition**( ) As Integer |
| cs | int **set_mountPosition**( MOUNTPOSITION **position**, MOUNTORIENTATION **orientation**) |
| java | int **set_mountPosition**( MOUNTPOSITION **position**, MOUNTORIENTATION **orientation**) |
| py | def **set_mountPosition**( **position**, **orientation**) |
| cmd | YRefFrame **target** **set_mountPosition** **position** **orientation** |

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

**Parameters :**

**position** a value among the `Y_MOUNTPOSITION` enumeration (`Y_MOUNTPOSITION_BOTTOM`, `Y_MOUNTPOSITION_TOP`, `Y_MOUNTPOSITION_FRONT`, `Y_MOUNTPOSITION_RIGHT`, `Y_MOUNTPOSITION_REAR`, `Y_MOUNTPOSITION_LEFT`), corresponding to the installation in a box, on one of the six faces.

**orientation** a value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`, `Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## refframe→set_userData() refframe→setUserData()[refframe setUserData: ] YRefFrame

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

### refframe→start3DCalibration()[refframe start3DCalibration] YRefFrame

Initiates the sensors tridimensional calibration process.

| | |
|---|---|
| js | function **start3DCalibration**( ) |
| nodejs | function **start3DCalibration**( ) |
| php | function **start3DCalibration**( ) |
| cpp | int **start3DCalibration**( ) |
| m | -(int) **start3DCalibration** |
| pas | function **start3DCalibration**( ): LongInt |
| vb | function **start3DCalibration**( ) As Integer |
| cs | int **start3DCalibration**( ) |
| java | int **start3DCalibration**( ) |
| py | def **start3DCalibration**( ) |
| cmd | YRefFrame **target** **start3DCalibration** |

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

## refframe→wait_async() YRefFrame

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.35. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_relay.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YRelay = yoctolib.YRelay; |
| php | require_once('yocto_relay.php'); |
| cpp | #include "yocto_relay.h" |
| m | #import "yocto_relay.h" |
| pas | uses yocto_relay; |
| vb | yocto_relay.vb |
| cs | yocto_relay.cs |
| java | import com.yoctopuce.YoctoAPI.YRelay; |
| py | from yocto_relay import * |

**Global functions**

**yFindRelay**(**func**)

Retrieves a relay for a given identifier.

**yFirstRelay**()

Starts the enumeration of relays currently accessible.

**`YRelay` methods**

**relay→delayedPulse**(**ms_delay**, **ms_duration**)

Schedules a pulse.

**relay→describe**()

Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**relay→get_advertisedValue**()

Returns the current value of the relay (no more than 6 characters).

**relay→get_countdown**()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

**relay→get_errorMessage**()

Returns the error message of the latest error with the relay.

**relay→get_errorType**()

Returns the numerical error code of the latest error with the relay.

**relay→get_friendlyName**()

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

**relay→get_functionDescriptor**()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**relay→get_functionId**()

Returns the hardware identifier of the relay, without reference to the module.

**relay→get_hardwareId**()

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

**relay→get_logicalName()**

Returns the logical name of the relay.

**relay→get_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

**relay→get_maxTimeOnStateB()**

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

**relay→get_module()**

Gets the `YModule` object for the device on which the function is located.

**relay→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**relay→get_output()**

Returns the output state of the relays, when used as a simple switch (single throw).

**relay→get_pulseTimer()**

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

**relay→get_state()**

Returns the state of the relays (A for the idle position, B for the active position).

**relay→get_stateAtPowerOn()**

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

**relay→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**relay→isOnline()**

Checks if the relay is currently reachable, without raising any error.

**relay→isOnline_async(callback, context)**

Checks if the relay is currently reachable, without raising any error (asynchronous version).

**relay→load(msValidity)**

Preloads the relay cache with a specified validity duration.

**relay→load_async(msValidity, callback, context)**

Preloads the relay cache with a specified validity duration (asynchronous version).

**relay→nextRelay()**

Continues the enumeration of relays started using `yFirstRelay()`.

**relay→pulse(ms_duration)**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

**relay→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**relay→set_logicalName(newval)**

Changes the logical name of the relay.

**relay→set_maxTimeOnStateA(newval)**

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

**relay→set_maxTimeOnStateB(newval)**

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

**relay→set_output(newval)**

Changes the output state of the relays, when used as a simple switch (single throw).

**relay→set_state(newval)**

Changes the state of the relays (A for the idle position, B for the active position).

**relay→set_stateAtPowerOn(newval)**

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

**relay→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**relay→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YRelay.FindRelay() yFindRelay()yFindRelay() YRelay

Retrieves a relay for a given identifier.

```
js      function yFindRelay( func)
nodejs  function FindRelay( func)
php     function yFindRelay( $func)
cpp     YRelay* yFindRelay( const string& func)
m       YRelay* yFindRelay( NSString* func)
pas     function yFindRelay( func: string): TYRelay
vb      function yFindRelay( ByVal func As String) As YRelay
cs      YRelay FindRelay( string func)
java    YRelay FindRelay( String func)
py      def FindRelay( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the relay is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRelay.isOnline()` to test if the relay is indeed online at a given time. In case of ambiguity when looking for a relay by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the relay

**Returns :**

a `YRelay` object allowing you to drive the relay.

## YRelay.FirstRelay() yFirstRelay()yFirstRelay() YRelay

Starts the enumeration of relays currently accessible.

| | |
|---|---|
| js | function **yFirstRelay**( ) |
| nodejs | function **FirstRelay**( ) |
| php | function **yFirstRelay**( ) |
| cpp | YRelay* **yFirstRelay**( ) |
| m | YRelay* **yFirstRelay**( ) |
| pas | function **yFirstRelay**( ): TYRelay |
| vb | function **yFirstRelay**( ) As YRelay |
| cs | YRelay **FirstRelay**( ) |
| java | YRelay **FirstRelay**( ) |
| py | def **FirstRelay**( ) |

Use the method `YRelay.nextRelay()` to iterate on next relays.

**Returns :**

a pointer to a `YRelay` object, corresponding to the first relay currently online, or a `null` pointer if there are none.

## relay→delayedPulse()[relay delayedPulse: ] YRelay

Schedules a pulse.

| | |
|---|---|
| js | function **delayedPulse**( **ms_delay**, **ms_duration**) |
| nodejs | function **delayedPulse**( **ms_delay**, **ms_duration**) |
| php | function **delayedPulse**( **$ms_delay**, **$ms_duration**) |
| cpp | int **delayedPulse**( int **ms_delay**, int **ms_duration**) |
| m | -(int) **delayedPulse** : (int) **ms_delay** : (int) **ms_duration** |
| pas | function **delayedPulse**( **ms_delay**: LongInt, **ms_duration**: LongInt): integer |
| vb | function **delayedPulse**( ByVal **ms_delay** As Integer,<br>ByVal **ms_duration** As Integer) As Integer |
| cs | int **delayedPulse**( int **ms_delay**, int **ms_duration**) |
| java | int **delayedPulse**( int **ms_delay**, int **ms_duration**) |
| py | def **delayedPulse**( **ms_delay**, **ms_duration**) |
| cmd | YRelay **target 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`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the relay (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## relay→get_advertisedValue() relay→advertisedValue()[relay advertisedValue] YRelay

Returns the current value of the relay (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YRelay **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the relay (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## relay→get_countdown() relay→countdown()[relay countdown] — YRelay

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

| | |
|---|---|
| js | function **get_countdown**( ) |
| nodejs | function **get_countdown**( ) |
| php | function **get_countdown**( ) |
| cpp | s64 **get_countdown**( ) |
| m | -(s64) countdown |
| pas | function **get_countdown**( ): int64 |
| vb | function **get_countdown**( ) As Long |
| cs | long **get_countdown**( ) |
| java | long **get_countdown**( ) |
| py | def **get_countdown**( ) |
| cmd | YRelay **target** **get_countdown** |

**Returns :**

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

## relay→get_errorMessage() relay→errorMessage()[relay errorMessage] YRelay

Returns the error message of the latest error with the relay.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the relay object

## relay→get_errorType() relay→errorType() YRelay

Returns the numerical error code of the latest error with the relay.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the relay object

## relay→get_friendlyName() relay→friendlyName()[relay friendlyName] YRelay

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the relay if they are defined, otherwise the serial number of the module and the hardware identifier of the relay (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the relay using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## relay→get_functionDescriptor() relay→functionDescriptor()[relay functionDescriptor] YRelay

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## relay→get_functionId() relay→functionId()[relay functionId] YRelay

Returns the hardware identifier of the relay, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the relay (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## relay→get_hardwareId() relay→hardwareId()[relay hardwareId] YRelay

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the relay. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the relay (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## relay→get_logicalName() relay→logicalName()[relay logicalName] YRelay

Returns the logical name of the relay.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YRelay **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the relay. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## relay→get_maxTimeOnStateA() relay→maxTimeOnStateA()[relay maxTimeOnStateA] YRelay

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

| | |
|---|---|
| js | function **get_maxTimeOnStateA**( ) |
| nodejs | function **get_maxTimeOnStateA**( ) |
| php | function **get_maxTimeOnStateA**( ) |
| cpp | s64 **get_maxTimeOnStateA**( ) |
| m | -(s64) maxTimeOnStateA |
| pas | function **get_maxTimeOnStateA**( ): int64 |
| vb | function **get_maxTimeOnStateA**( ) As Long |
| cs | long **get_maxTimeOnStateA**( ) |
| java | long **get_maxTimeOnStateA**( ) |
| py | def **get_maxTimeOnStateA**( ) |
| cmd | YRelay **target** **get_maxTimeOnStateA** |

Zero means no maximum time.

**Returns :**

an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

## relay→get_maxTimeOnStateB() relay→maxTimeOnStateB()[relay maxTimeOnStateB] YRelay

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

| | |
|---|---|
| js | function **get_maxTimeOnStateB**( ) |
| nodejs | function **get_maxTimeOnStateB**( ) |
| php | function **get_maxTimeOnStateB**( ) |
| cpp | s64 **get_maxTimeOnStateB**( ) |
| m | -(s64) maxTimeOnStateB |
| pas | function **get_maxTimeOnStateB**( ): int64 |
| vb | function **get_maxTimeOnStateB**( ) As Long |
| cs | long **get_maxTimeOnStateB**( ) |
| java | long **get_maxTimeOnStateB**( ) |
| py | def **get_maxTimeOnStateB**( ) |
| cmd | YRelay **target** **get_maxTimeOnStateB** |

Zero means no maximum time.

**Returns :**
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

## relay→get_module() relay→module()[relay module] YRelay

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## relay→get_module_async() relay→module_async() **YRelay**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## relay→get_output() relay→output()[relay output] **YRelay**

Returns the output state of the relays, when used as a simple switch (single throw).

| | |
|---|---|
| js | function **get_output**( ) |
| nodejs | function **get_output**( ) |
| php | function **get_output**( ) |
| cpp | Y_OUTPUT_enum **get_output**( ) |
| m | -(Y_OUTPUT_enum) output |
| pas | function **get_output**( ): Integer |
| vb | function **get_output**( ) As Integer |
| cs | int **get_output**( ) |
| java | int **get_output**( ) |
| py | def **get_output**( ) |
| cmd | YRelay **target** **get_output** |

**Returns :**

either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns `Y_OUTPUT_INVALID`.

## relay→get_pulseTimer() relay→pulseTimer()[relay pulseTimer] YRelay

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

| | |
|---|---|
| js | function **get_pulseTimer**( ) |
| nodejs | function **get_pulseTimer**( ) |
| php | function **get_pulseTimer**( ) |
| cpp | s64 **get_pulseTimer**( ) |
| m | -(s64) pulseTimer |
| pas | function **get_pulseTimer**( ): int64 |
| vb | function **get_pulseTimer**( ) As Long |
| cs | long **get_pulseTimer**( ) |
| java | long **get_pulseTimer**( ) |
| py | def **get_pulseTimer**( ) |
| cmd | YRelay **target** **get_pulseTimer** |

When there is no ongoing pulse, returns zero.

**Returns :**
an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

## relay→get_state() relay→state()[relay state] YRelay

Returns the state of the relays (A for the idle position, B for the active position).

| | |
|---|---|
| js | function **get_state**( ) |
| nodejs | function **get_state**( ) |
| php | function **get_state**( ) |
| cpp | Y_STATE_enum **get_state**( ) |
| m | -(Y_STATE_enum) state |
| pas | function **get_state**( ): Integer |
| vb | function **get_state**( ) As Integer |
| cs | int **get_state**( ) |
| java | int **get_state**( ) |
| py | def **get_state**( ) |
| cmd | YRelay **target** **get_state** |

**Returns :**

either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

## relay→get_stateAtPowerOn() relay→stateAtPowerOn()[relay stateAtPowerOn] YRelay

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

| | |
|---|---|
| js | function **get_stateAtPowerOn**( ) |
| nodejs | function **get_stateAtPowerOn**( ) |
| php | function **get_stateAtPowerOn**( ) |
| cpp | Y_STATEATPOWERON_enum **get_stateAtPowerOn**( ) |
| m | -(Y_STATEATPOWERON_enum) stateAtPowerOn |
| pas | function **get_stateAtPowerOn**( ): Integer |
| vb | function **get_stateAtPowerOn**( ) As Integer |
| cs | int **get_stateAtPowerOn**( ) |
| java | int **get_stateAtPowerOn**( ) |
| py | def **get_stateAtPowerOn**( ) |
| cmd | YRelay **target** **get_stateAtPowerOn** |

**Returns :**

a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B` corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns `Y_STATEATPOWERON_INVALID`.

## relay→get_userData() relay→userData()[relay userData] YRelay

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## relay→isOnline()[relay isOnline] YRelay

Checks if the relay is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the relay.

**Returns :**

`true` if the relay can be reached, and `false` otherwise

## relay→isOnline_async() YRelay

Checks if the relay is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## relay→load()[relay load: ] YRelay

Preloads the relay cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## relay→load_async() YRelay

Preloads the relay cache with a specified validity duration (asynchronous version).

js function **load_async**( **msValidity**, **callback**, **context**)
nodejs function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## relay→nextRelay()[relay nextRelay] YRelay

Continues the enumeration of relays started using `yFirstRelay()`.

| | |
|---|---|
| js | function **nextRelay**( ) |
| nodejs | function **nextRelay**( ) |
| php | function **nextRelay**( ) |
| cpp | YRelay * **nextRelay**( ) |
| m | -(YRelay*) **nextRelay** |
| pas | function **nextRelay**( ): TYRelay |
| vb | function **nextRelay**( ) As YRelay |
| cs | YRelay **nextRelay**( ) |
| java | YRelay **nextRelay**( ) |
| py | def **nextRelay**( ) |

**Returns :**

a pointer to a `YRelay` object, corresponding to a relay currently online, or a `null` pointer if there are no more relays to enumerate.

## relay→pulse()[relay pulse: ] YRelay

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

| | |
|---|---|
| js | function **pulse**( **ms_duration**) |
| nodejs | function **pulse**( **ms_duration**) |
| php | function **pulse**( **$ms_duration**) |
| cpp | int **pulse**( int **ms_duration**) |
| m | -(int) **pulse** : (int) **ms_duration** |
| pas | function **pulse**( **ms_duration**: LongInt): integer |
| vb | function **pulse**( ByVal **ms_duration** As Integer) As Integer |
| cs | int **pulse**( int **ms_duration**) |
| java | int **pulse**( int **ms_duration**) |
| py | def **pulse**( **ms_duration**) |
| cmd | YRelay **target** **pulse** **ms_duration** |

**Parameters :**

**ms_duration** pulse duration, in millisecondes

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→registerValueCallback()[relay registerValueCallback: ] YRelay

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YRelayValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YRelayValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYRelayValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## relay→set_logicalName() relay→setLogicalName()[relay setLogicalName: ] YRelay

Changes the logical name of the relay.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YRelay **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the relay.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## relay→set_maxTimeOnStateA() relay→setMaxTimeOnStateA()[relay setMaxTimeOnStateA: ] YRelay

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

| | |
|---|---|
| js | function **set_maxTimeOnStateA**( **newval**) |
| nodejs | function **set_maxTimeOnStateA**( **newval**) |
| php | function **set_maxTimeOnStateA**( **$newval**) |
| cpp | int **set_maxTimeOnStateA**( s64 **newval**) |
| m | -(int) setMaxTimeOnStateA : (s64) **newval** |
| pas | function **set_maxTimeOnStateA**( **newval**: int64): integer |
| vb | function **set_maxTimeOnStateA**( ByVal **newval** As Long) As Integer |
| cs | int **set_maxTimeOnStateA**( long **newval**) |
| java | int **set_maxTimeOnStateA**( long **newval**) |
| py | def **set_maxTimeOnStateA**( **newval**) |
| cmd | YRelay **target** **set_maxTimeOnStateA** **newval** |

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→set_maxTimeOnStateB() relay→setMaxTimeOnStateB()[relay setMaxTimeOnStateB: ] YRelay

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

| | |
|---|---|
| `js` | function **set_maxTimeOnStateB**( **newval**) |
| `nodejs` | function **set_maxTimeOnStateB**( **newval**) |
| `php` | function **set_maxTimeOnStateB**( **$newval**) |
| `cpp` | int **set_maxTimeOnStateB**( s64 **newval**) |
| `m` | -(int) setMaxTimeOnStateB : (s64) **newval** |
| `pas` | function **set_maxTimeOnStateB**( **newval**: int64): integer |
| `vb` | function **set_maxTimeOnStateB**( ByVal **newval** As Long) As Integer |
| `cs` | int **set_maxTimeOnStateB**( long **newval**) |
| `java` | int **set_maxTimeOnStateB**( long **newval**) |
| `py` | def **set_maxTimeOnStateB**( **newval**) |
| `cmd` | YRelay **target** **set_maxTimeOnStateB** **newval** |

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→set_output() relay→setOutput()[relay setOutput: ] YRelay

Changes the output state of the relays, when used as a simple switch (single throw).

| | |
|---|---|
| js | function **set_output**( **newval**) |
| nodejs | function **set_output**( **newval**) |
| php | function **set_output**( **$newval**) |
| cpp | int **set_output**( Y_OUTPUT_enum **newval**) |
| m | -(int) setOutput : (Y_OUTPUT_enum) **newval** |
| pas | function **set_output**( **newval**: Integer): integer |
| vb | function **set_output**( ByVal **newval** As Integer) As Integer |
| cs | int **set_output**( int **newval**) |
| java | int **set_output**( int **newval**) |
| py | def **set_output**( **newval**) |
| cmd | YRelay **target set_output newval** |

**Parameters :**

**newval** either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the relays, when used as a simple switch (single throw)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→set_state() relay→setState()[relay setState: ] YRelay

Changes the state of the relays (A for the idle position, B for the active position).

| | |
|---|---|
| js | function **set_state**( **newval**) |
| nodejs | function **set_state**( **newval**) |
| php | function **set_state**( **$newval**) |
| cpp | int **set_state**( Y_STATE_enum **newval**) |
| m | -(int) setState : (Y_STATE_enum) **newval** |
| pas | function **set_state**( **newval**: Integer): integer |
| vb | function **set_state**( ByVal **newval** As Integer) As Integer |
| cs | int **set_state**( int **newval**) |
| java | int **set_state**( int **newval**) |
| py | def **set_state**( **newval**) |
| cmd | YRelay **target** **set_state** **newval** |

**Parameters :**

**newval** either `Y_STATE_A` or `Y_STATE_B`, according to the state of the relays (A for the idle position, B for the active position)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→set_stateAtPowerOn() relay→setStateAtPowerOn()[relay setStateAtPowerOn: ] — YRelay

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

| | |
|---|---|
| js | function **set_stateAtPowerOn**( **newval**) |
| nodejs | function **set_stateAtPowerOn**( **newval**) |
| php | function **set_stateAtPowerOn**( **$newval**) |
| cpp | int **set_stateAtPowerOn**( Y_STATEATPOWERON_enum **newval**) |
| m | -(int) setStateAtPowerOn : (Y_STATEATPOWERON_enum) **newval** |
| pas | function **set_stateAtPowerOn**( **newval**: Integer): integer |
| vb | function **set_stateAtPowerOn**( ByVal **newval** As Integer) As Integer |
| cs | int **set_stateAtPowerOn**( int **newval**) |
| java | int **set_stateAtPowerOn**( int **newval**) |
| py | def **set_stateAtPowerOn**( **newval**) |
| cmd | YRelay **target** **set_stateAtPowerOn** **newval** |

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→set_userData() relay→setUserData()[relay setUserData: ] YRelay

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## relay→wait_async() YRelay

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.36. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>var YModule = yoctolib.YModule; |
| php | require_once('yocto_api.php'); |
| cpp | #include "yocto_api.h" |
| m | #import "yocto_api.h" |
| pas | uses yocto_api; |
| vb | yocto_api.vb |
| cs | yocto_api.cs |
| java | import com.yoctopuce.YoctoAPI.YModule; |
| py | from yocto_api import * |

| Global functions |
|---|
| **yFindSensor(func)**<br>Retrieves a sensor for a given identifier. |
| **yFirstSensor()**<br>Starts the enumeration of sensors currently accessible. |
| **YSensor methods** |
| **sensor→calibrateFromPoints(rawValues, refValues)**<br>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure. |
| **sensor→describe()**<br>Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **sensor→get_advertisedValue()**<br>Returns the current value of the sensor (no more than 6 characters). |
| **sensor→get_currentRawValue()**<br>Returns the uncalibrated, unrounded raw value returned by the sensor. |
| **sensor→get_currentValue()**<br>Returns the current value of the measure. |
| **sensor→get_errorMessage()**<br>Returns the error message of the latest error with the sensor. |
| **sensor→get_errorType()**<br>Returns the numerical error code of the latest error with the sensor. |
| **sensor→get_friendlyName()**<br>Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`. |
| **sensor→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **sensor→get_functionId()**<br>Returns the hardware identifier of the sensor, without reference to the module. |
| **sensor→get_hardwareId()** |

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

**sensor→get_highestValue()**

Returns the maximal value observed for the measure since the device was started.

**sensor→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**sensor→get_logicalName()**

Returns the logical name of the sensor.

**sensor→get_lowestValue()**

Returns the minimal value observed for the measure since the device was started.

**sensor→get_module()**

Gets the `YModule` object for the device on which the function is located.

**sensor→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**sensor→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**sensor→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**sensor→get_resolution()**

Returns the resolution of the measured values.

**sensor→get_unit()**

Returns the measuring unit for the measure.

**sensor→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**sensor→isOnline()**

Checks if the sensor is currently reachable, without raising any error.

**sensor→isOnline_async(callback, context)**

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

**sensor→load(msValidity)**

Preloads the sensor cache with a specified validity duration.

**sensor→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**sensor→load_async(msValidity, callback, context)**

Preloads the sensor cache with a specified validity duration (asynchronous version).

**sensor→nextSensor()**

Continues the enumeration of sensors started using `yFirstSensor()`.

**sensor→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**sensor→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**sensor→set_highestValue(newval)**

Changes the recorded maximal value observed.

**sensor→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**sensor→set_logicalName(newval)**

Changes the logical name of the sensor.

**sensor→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**sensor→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**sensor→set_resolution(newval)**

Changes the resolution of the measured physical values.

**sensor→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**sensor→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YSensor.FindSensor() yFindSensor()yFindSensor() YSensor

Retrieves a sensor for a given identifier.

| | |
|---|---|
| js | function **yFindSensor**( **func**) |
| nodejs | function **FindSensor**( **func**) |
| php | function **yFindSensor**( **$func**) |
| cpp | YSensor* **yFindSensor**( string **func**) |
| m | +(YSensor*) **yFindSensor** : (NSString*) **func** |
| pas | function **yFindSensor**( **func**: string): TYSensor |
| vb | function **yFindSensor**( ByVal **func** As String) As YSensor |
| cs | YSensor **FindSensor**( string **func**) |
| java | YSensor **FindSensor**( String **func**) |
| py | def **FindSensor**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the sensor

**Returns :**

a `YSensor` object allowing you to drive the sensor.

## YSensor.FirstSensor() yFirstSensor()yFirstSensor() YSensor

Starts the enumeration of sensors currently accessible.

| | |
|---|---|
| js | function **yFirstSensor**( ) |
| nodejs | function **FirstSensor**( ) |
| php | function **yFirstSensor**( ) |
| cpp | YSensor* **yFirstSensor**( ) |
| m | YSensor* **yFirstSensor**( ) |
| pas | function **yFirstSensor**( ): TYSensor |
| vb | function **yFirstSensor**( ) As YSensor |
| cs | YSensor **FirstSensor**( ) |
| java | YSensor **FirstSensor**( ) |
| py | def **FirstSensor**( ) |

Use the method `YSensor.nextSensor()` to iterate on next sensors.

**Returns :**

a pointer to a `YSensor` object, corresponding to the first sensor currently online, or a `null` pointer if there are none.

## sensor→calibrateFromPoints()[sensor calibrateFromPoints: ] YSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YSensor **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→describe()[sensor describe] YSensor

Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## sensor→get_advertisedValue() sensor→advertisedValue()[sensor advertisedValue] YSensor

Returns the current value of the sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YSensor **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## sensor→get_currentRawValue() sensor→currentRawValue()[sensor currentRawValue] YSensor

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YSensor **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## sensor→get_currentValue() sensor→currentValue()[sensor currentValue]

**YSensor**

Returns the current value of the measure.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YSensor **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current value of the measure

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## sensor→get_errorMessage() sensor→errorMessage()[sensor errorMessage] YSensor

Returns the error message of the latest error with the sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the sensor object

## sensor→get_errorType() sensor→errorType() YSensor

Returns the numerical error code of the latest error with the sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the sensor object

## sensor→get_friendlyName() YSensor
## sensor→friendlyName()[sensor friendlyName]

Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## sensor→get_functionDescriptor() sensor→functionDescriptor()[sensor functionDescriptor]

**YSensor**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## sensor→get_functionId() sensor→functionId()[sensor functionId] YSensor

Returns the hardware identifier of the sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## sensor→get_hardwareId() sensor→hardwareId()[sensor hardwareId] — YSensor

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## sensor→get_highestValue()<br>sensor→highestValue()[sensor highestValue] — YSensor

Returns the maximal value observed for the measure since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YSensor **target** **get_highestValue** |

**Returns :**
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## sensor→get_logFrequency() sensor→logFrequency()[sensor logFrequency] YSensor

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YSensor **target** **get_logFrequency** |

**Returns :**
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## sensor→get_logicalName() sensor→logicalName()[sensor logicalName] YSensor

Returns the logical name of the sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YSensor **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## sensor→get_lowestValue() sensor→lowestValue()[sensor lowestValue] YSensor

Returns the minimal value observed for the measure since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YSensor **target** **get_lowestValue** |

**Returns :**

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## sensor→get_module() sensor→module()[sensor module] YSensor

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## sensor→get_module_async() sensor→module_async() — YSensor

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## sensor→get_recordedData() sensor→recordedData()[sensor recordedData: ] YSensor

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YSensor **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## sensor→get_reportFrequency() sensor→reportFrequency()[sensor reportFrequency] YSensor

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YSensor **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## sensor→get_resolution() sensor→resolution()[sensor resolution] YSensor

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YSensor **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## sensor→get_unit() sensor→unit()[sensor unit]

**YSensor**

Returns the measuring unit for the measure.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YSensor **target** **get_unit** |

**Returns :**

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## sensor→get_userData() sensor→userData()[sensor userData] YSensor

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## sensor→isOnline()[sensor isOnline] YSensor

Checks if the sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the sensor.

**Returns :**

`true` if the sensor can be reached, and `false` otherwise

## sensor→isOnline_async() YSensor

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## sensor→load()[sensor load: ] YSensor

Preloads the sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## sensor→loadCalibrationPoints()[sensor loadCalibrationPoints: ] YSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YSensor **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→load_async() YSensor

Preloads the sensor cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## sensor→nextSensor()[sensor nextSensor] YSensor

Continues the enumeration of sensors started using `yFirstSensor()`.

| | |
|---|---|
| js | function **nextSensor**( ) |
| nodejs | function **nextSensor**( ) |
| php | function **nextSensor**( ) |
| cpp | YSensor * **nextSensor**( ) |
| m | -(YSensor*) **nextSensor** |
| pas | function **nextSensor**( ): TYSensor |
| vb | function **nextSensor**( ) As YSensor |
| cs | YSensor **nextSensor**( ) |
| java | YSensor **nextSensor**( ) |
| py | def **nextSensor**( ) |

**Returns :**

a pointer to a `YSensor` object, corresponding to a sensor currently online, or a `null` pointer if there are no more sensors to enumerate.

## sensor→registerTimedReportCallback()[sensor registerTimedReportCallback: ] YSensor

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YSensorTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YSensorTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYSensorTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## sensor→registerValueCallback()[sensor registerValueCallback: ] YSensor

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YSensorValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YSensorValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYSensorValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## sensor→set_highestValue() sensor→setHighestValue()[sensor setHighestValue: ] YSensor

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YSensor **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→set_logFrequency() sensor→setLogFrequency()[sensor setLogFrequency: ] YSensor

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YSensor **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→set_logicalName() sensor→setLogicalName()[sensor setLogicalName: ] YSensor

Changes the logical name of the sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YSensor **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## sensor→set_lowestValue() sensor→setLowestValue()[sensor setLowestValue: ] YSensor

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YSensor **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→set_reportFrequency() sensor→setReportFrequency()[sensor setReportFrequency: ] YSensor

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YSensor **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→set_resolution() sensor→setResolution()[sensor setResolution: ] YSensor

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YSensor **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## sensor→set_userData() sensor→setUserData()[sensor setUserData: ] YSensor

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## sensor→wait_async() YSensor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

`js` function **wait_async**( **callback**, **context**)
`nodejs` function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.37. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_servo.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YServo = yoctolib.YServo; |
| php | require_once('yocto_servo.php'); |
| cpp | #include "yocto_servo.h" |
| m | #import "yocto_servo.h" |
| pas | uses yocto_servo; |
| vb | yocto_servo.vb |
| cs | yocto_servo.cs |
| java | import com.yoctopuce.YoctoAPI.YServo; |
| py | from yocto_servo import * |

**Global functions**

**yFindServo**(**func**)
Retrieves a servo for a given identifier.

**yFirstServo**()
Starts the enumeration of servos currently accessible.

**YServo methods**

**servo→describe**()
Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**servo→get_advertisedValue**()
Returns the current value of the servo (no more than 6 characters).

**servo→get_enabled**()
Returns the state of the servos.

**servo→get_enabledAtPowerOn**()
Returns the servo signal generator state at power up.

**servo→get_errorMessage**()
Returns the error message of the latest error with the servo.

**servo→get_errorType**()
Returns the numerical error code of the latest error with the servo.

**servo→get_friendlyName**()
Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.

**servo→get_functionDescriptor**()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**servo→get_functionId**()
Returns the hardware identifier of the servo, without reference to the module.

**servo→get_hardwareId**()
Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.

**servo→get_logicalName**()
Returns the logical name of the servo.

**servo→get_module()**

Gets the `YModule` object for the device on which the function is located.

**servo→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**servo→get_neutral()**

Returns the duration in microseconds of a neutral pulse for the servo.

**servo→get_position()**

Returns the current servo position.

**servo→get_positionAtPowerOn()**

Returns the servo position at device power up.

**servo→get_range()**

Returns the current range of use of the servo.

**servo→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**servo→isOnline()**

Checks if the servo is currently reachable, without raising any error.

**servo→isOnline_async(callback, context)**

Checks if the servo is currently reachable, without raising any error (asynchronous version).

**servo→load(msValidity)**

Preloads the servo cache with a specified validity duration.

**servo→load_async(msValidity, callback, context)**

Preloads the servo cache with a specified validity duration (asynchronous version).

**servo→move(target, ms_duration)**

Performs a smooth move at constant speed toward a given position.

**servo→nextServo()**

Continues the enumeration of servos started using `yFirstServo()`.

**servo→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**servo→set_enabled(newval)**

Stops or starts the servo.

**servo→set_enabledAtPowerOn(newval)**

Configure the servo signal generator state at power up.

**servo→set_logicalName(newval)**

Changes the logical name of the servo.

**servo→set_neutral(newval)**

Changes the duration of the pulse corresponding to the neutral position of the servo.

**servo→set_position(newval)**

Changes immediately the servo driving position.

**servo→set_positionAtPowerOn(newval)**

Configure the servo position at device power up.

**servo→set_range(newval)**

Changes the range of use of the servo, specified in per cents.

**servo→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**servo→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YServo.FindServo() yFindServo()yFindServo() YServo

Retrieves a servo for a given identifier.

| | |
|---|---|
| js | function **yFindServo**( **func**) |
| nodejs | function **FindServo**( **func**) |
| php | function **yFindServo**( **$func**) |
| cpp | YServo* **yFindServo**( const string& **func**) |
| m | YServo* **yFindServo**( NSString* **func**) |
| pas | function **yFindServo**( **func**: string): TYServo |
| vb | function **yFindServo**( ByVal **func** As String) As YServo |
| cs | YServo **FindServo**( string **func**) |
| java | YServo **FindServo**( String **func**) |
| py | def **FindServo**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the servo is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YServo.isOnline()` to test if the servo is indeed online at a given time. In case of ambiguity when looking for a servo by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the servo

**Returns :**

a `YServo` object allowing you to drive the servo.

## YServo.FirstServo() yFirstServo()yFirstServo() YServo

Starts the enumeration of servos currently accessible.

| | |
|---|---|
| js | function **yFirstServo**( ) |
| nodejs | function **FirstServo**( ) |
| php | function **yFirstServo**( ) |
| cpp | YServo* **yFirstServo**( ) |
| m | YServo* **yFirstServo**( ) |
| pas | function **yFirstServo**( ): TYServo |
| vb | function **yFirstServo**( ) As YServo |
| cs | YServo **FirstServo**( ) |
| java | YServo **FirstServo**( ) |
| py | def **FirstServo**( ) |

Use the method `YServo.nextServo()` to iterate on next servos.

**Returns :**

a pointer to a `YServo` object, corresponding to the first servo currently online, or a `null` pointer if there are none.

## servo→describe()[servo describe] YServo

Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the servo (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## servo→get_advertisedValue() servo→advertisedValue()[servo advertisedValue] YServo

Returns the current value of the servo (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YServo **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the servo (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## servo→get_enabled() servo→enabled()[servo enabled] YServo

Returns the state of the servos.

| | |
|---|---|
| js | function **get_enabled**( ) |
| nodejs | function **get_enabled**( ) |
| php | function **get_enabled**( ) |
| cpp | Y_ENABLED_enum **get_enabled**( ) |
| m | -(Y_ENABLED_enum) enabled |
| pas | function **get_enabled**( ): Integer |
| vb | function **get_enabled**( ) As Integer |
| cs | int **get_enabled**( ) |
| java | int **get_enabled**( ) |
| py | def **get_enabled**( ) |
| cmd | YServo **target** **get_enabled** |

**Returns :**

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the state of the servos

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

## servo→get_enabledAtPowerOn() servo→enabledAtPowerOn()[servo enabledAtPowerOn] YServo

Returns the servo signal generator state at power up.

| | |
|---|---|
| js | function **get_enabledAtPowerOn**( ) |
| nodejs | function **get_enabledAtPowerOn**( ) |
| php | function **get_enabledAtPowerOn**( ) |
| cpp | Y_ENABLEDATPOWERON_enum **get_enabledAtPowerOn**( ) |
| m | -(Y_ENABLEDATPOWERON_enum) enabledAtPowerOn |
| pas | function **get_enabledAtPowerOn**( ): Integer |
| vb | function **get_enabledAtPowerOn**( ) As Integer |
| cs | int **get_enabledAtPowerOn**( ) |
| java | int **get_enabledAtPowerOn**( ) |
| py | def **get_enabledAtPowerOn**( ) |
| cmd | YServo **target** **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()<br>servo→errorMessage()[servo errorMessage] YServo

Returns the error message of the latest error with the servo.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the servo object

## servo→get_errorType() servo→errorType() YServo

Returns the numerical error code of the latest error with the servo.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the servo object

## servo→get_friendlyName() servo→friendlyName()[servo friendlyName] — YServo

Returns a global identifier of the servo in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the servo if they are defined, otherwise the serial number of the module and the hardware identifier of the servo (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the servo using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## servo→get_functionDescriptor() servo→functionDescriptor()[servo functionDescriptor] YServo

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## servo→get_functionId() servo→functionId()[servo functionId] YServo

Returns the hardware identifier of the servo, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the servo (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## servo→get_hardwareId() servo→hardwareId()[servo hardwareId] YServo

Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the servo. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the servo (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## servo→get_logicalName() servo→logicalName()[servo logicalName] YServo

Returns the logical name of the servo.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YServo **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the servo. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## servo→get_module() servo→module()[servo module] YServo

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## servo→get_module_async() servo→module_async() **YServo**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`js` function **get_module_async**( **callback**, **context**)

`nodejs` function **get_module_async**( **callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## servo→get_neutral() servo→neutral()[servo neutral] YServo

Returns the duration in microseconds of a neutral pulse for the servo.

| | |
|---|---|
| js | function **get_neutral**( ) |
| nodejs | function **get_neutral**( ) |
| php | function **get_neutral**( ) |
| cpp | int **get_neutral**( ) |
| m | -(int) neutral |
| pas | function **get_neutral**( ): LongInt |
| vb | function **get_neutral**( ) As Integer |
| cs | int **get_neutral**( ) |
| java | int **get_neutral**( ) |
| py | def **get_neutral**( ) |
| cmd | YServo **target** **get_neutral** |

**Returns :**
an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns `Y_NEUTRAL_INVALID`.

## servo→get_position() servo→position()[servo position] YServo

Returns the current servo position.

| | |
|---|---|
| js | function **get_position**( ) |
| nodejs | function **get_position**( ) |
| php | function **get_position**( ) |
| cpp | int **get_position**( ) |
| m | -(int) position |
| pas | function **get_position**( ): LongInt |
| vb | function **get_position**( ) As Integer |
| cs | int **get_position**( ) |
| java | int **get_position**( ) |
| py | def **get_position**( ) |
| cmd | YServo **target** **get_position** |

**Returns :**
an integer corresponding to the current servo position

On failure, throws an exception or returns `Y_POSITION_INVALID`.

## servo→get_positionAtPowerOn() servo→positionAtPowerOn()[servo positionAtPowerOn] YServo

Returns the servo position at device power up.

| | |
|---|---|
| js | function **get_positionAtPowerOn**( ) |
| nodejs | function **get_positionAtPowerOn**( ) |
| php | function **get_positionAtPowerOn**( ) |
| cpp | int **get_positionAtPowerOn**( ) |
| m | -(int) positionAtPowerOn |
| pas | function **get_positionAtPowerOn**( ): LongInt |
| vb | function **get_positionAtPowerOn**( ) As Integer |
| cs | int **get_positionAtPowerOn**( ) |
| java | int **get_positionAtPowerOn**( ) |
| py | def **get_positionAtPowerOn**( ) |
| cmd | YServo **target** **get_positionAtPowerOn** |

**Returns :**
an integer corresponding to the servo position at device power up

On failure, throws an exception or returns `Y_POSITIONATPOWERON_INVALID`.

## servo→get_range() servo→range()[servo range] YServo

Returns the current range of use of the servo.

| | |
|---|---|
| js | function **get_range**( ) |
| nodejs | function **get_range**( ) |
| php | function **get_range**( ) |
| cpp | int **get_range**( ) |
| m | -(int) range |
| pas | function **get_range**( ): LongInt |
| vb | function **get_range**( ) As Integer |
| cs | int **get_range**( ) |
| java | int **get_range**( ) |
| py | def **get_range**( ) |
| cmd | YServo **target** **get_range** |

**Returns :**
an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns `Y_RANGE_INVALID`.

## servo→get_userData() servo→userData()[servo userData] YServo

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## servo→isOnline()[servo isOnline] YServo

Checks if the servo is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

**Returns :**

`true` if the servo can be reached, and `false` otherwise

## servo→isOnline_async() YServo

Checks if the servo is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## servo→load()[servo load: ] YServo

Preloads the servo cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## servo→load_async() YServo

Preloads the servo cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## servo→move()[servo move: ] YServo

Performs a smooth move at constant speed toward a given position.

| | |
|---|---|
| js | function **move**( **target**, **ms_duration**) |
| nodejs | function **move**( **target**, **ms_duration**) |
| php | function **move**( **$target**, **$ms_duration**) |
| cpp | int **move**( int **target**, int **ms_duration**) |
| m | -(int) **move** : (int) **target** : (int) **ms_duration** |
| pas | function **move**( **target**: LongInt, **ms_duration**: LongInt): integer |
| vb | function **move**( ByVal **target** As Integer,<br>ByVal **ms_duration** As Integer) As Integer |
| cs | int **move**( int **target**, int **ms_duration**) |
| java | int **move**( int **target**, int **ms_duration**) |
| py | def **move**( **target**, **ms_duration**) |
| cmd | YServo **target** **move** **target** **ms_duration** |

**Parameters :**

**target** new position at the end of the move

**ms_duration** total duration of the move, in milliseconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→nextServo()[servo nextServo] YServo

Continues the enumeration of servos started using `yFirstServo()`.

| | |
|---|---|
| `js` | function **nextServo**( ) |
| `nodejs` | function **nextServo**( ) |
| `php` | function **nextServo**( ) |
| `cpp` | YServo * **nextServo**( ) |
| `m` | -(YServo*) **nextServo** |
| `pas` | function **nextServo**( ): TYServo |
| `vb` | function **nextServo**( ) As YServo |
| `cs` | YServo **nextServo**( ) |
| `java` | YServo **nextServo**( ) |
| `py` | def **nextServo**( ) |

**Returns :**

a pointer to a `YServo` object, corresponding to a servo currently online, or a `null` pointer if there are no more servos to enumerate.

## servo→registerValueCallback()[servo registerValueCallback: ] YServo

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YServoValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YServoValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYServoValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## servo→set_enabled() servo→setEnabled()[servo setEnabled: ] YServo

Stops or starts the servo.

| | |
|---|---|
| js | function **set_enabled**( **newval**) |
| nodejs | function **set_enabled**( **newval**) |
| php | function **set_enabled**( **$newval**) |
| cpp | int **set_enabled**( Y_ENABLED_enum **newval**) |
| m | -(int) setEnabled : (Y_ENABLED_enum) **newval** |
| pas | function **set_enabled**( **newval**: Integer): integer |
| vb | function **set_enabled**( ByVal **newval** As Integer) As Integer |
| cs | int **set_enabled**( int **newval**) |
| java | int **set_enabled**( int **newval**) |
| py | def **set_enabled**( **newval**) |
| cmd | YServo **target set_enabled newval** |

**Parameters :**

**newval** either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→set_enabledAtPowerOn() servo→setEnabledAtPowerOn()[servo setEnabledAtPowerOn: ] YServo

Configure the servo signal generator state at power up.

| | |
|---|---|
| js | function **set_enabledAtPowerOn**( **newval**) |
| nodejs | function **set_enabledAtPowerOn**( **newval**) |
| php | function **set_enabledAtPowerOn**( **$newval**) |
| cpp | int **set_enabledAtPowerOn**( Y_ENABLEDATPOWERON_enum **newval**) |
| m | -(int) setEnabledAtPowerOn : (Y_ENABLEDATPOWERON_enum) **newval** |
| pas | function **set_enabledAtPowerOn**( **newval**: Integer): integer |
| vb | function **set_enabledAtPowerOn**( ByVal **newval** As Integer) As Integer |
| cs | int **set_enabledAtPowerOn**( int **newval**) |
| java | int **set_enabledAtPowerOn**( int **newval**) |
| py | def **set_enabledAtPowerOn**( **newval**) |
| cmd | YServo **target** **set_enabledAtPowerOn** **newval** |

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→set_logicalName() servo→setLogicalName()[servo setLogicalName: ] YServo

Changes the logical name of the servo.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YServo **target set_logicalName newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the servo.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## servo→set_neutral() servo→setNeutral()[servo setNeutral: ] YServo

Changes the duration of the pulse corresponding to the neutral position of the servo.

| | |
|---|---|
| js | function **set_neutral**( **newval**) |
| nodejs | function **set_neutral**( **newval**) |
| php | function **set_neutral**( **$newval**) |
| cpp | int **set_neutral**( int **newval**) |
| m | -(int) setNeutral : (int) **newval** |
| pas | function **set_neutral**( **newval**: LongInt): integer |
| vb | function **set_neutral**( ByVal **newval** As Integer) As Integer |
| cs | int **set_neutral**( int **newval**) |
| java | int **set_neutral**( int **newval**) |
| py | def **set_neutral**( **newval**) |
| cmd | YServo **target** **set_neutral** **newval** |

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

**Parameters :**

**newval** an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→set_position() servo→setPosition()[servo setPosition: ] YServo

Changes immediately the servo driving position.

| | |
|---|---|
| js | function **set_position**( **newval**) |
| nodejs | function **set_position**( **newval**) |
| php | function **set_position**( **$newval**) |
| cpp | int **set_position**( int **newval**) |
| m | -(int) setPosition : (int) **newval** |
| pas | function **set_position**( **newval**: LongInt): integer |
| vb | function **set_position**( ByVal **newval** As Integer) As Integer |
| cs | int **set_position**( int **newval**) |
| java | int **set_position**( int **newval**) |
| py | def **set_position**( **newval**) |
| cmd | YServo **target** **set_position** **newval** |

**Parameters :**

**newval** an integer corresponding to immediately the servo driving position

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→set_positionAtPowerOn() servo→setPositionAtPowerOn()[servo setPositionAtPowerOn: ] YServo

Configure the servo position at device power up.

| | |
|---|---|
| js | function **set_positionAtPowerOn**( **newval**) |
| nodejs | function **set_positionAtPowerOn**( **newval**) |
| php | function **set_positionAtPowerOn**( **$newval**) |
| cpp | int **set_positionAtPowerOn**( int **newval**) |
| m | -(int) setPositionAtPowerOn : (int) **newval** |
| pas | function **set_positionAtPowerOn**( **newval**: LongInt): integer |
| vb | function **set_positionAtPowerOn**( ByVal **newval** As Integer) As Integer |
| cs | int **set_positionAtPowerOn**( int **newval**) |
| java | int **set_positionAtPowerOn**( int **newval**) |
| py | def **set_positionAtPowerOn**( **newval**) |
| cmd | YServo **target** **set_positionAtPowerOn** **newval** |

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** an integer

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→set_range() servo→setRange()[servo setRange: ] YServo

Changes the range of use of the servo, specified in per cents.

| | |
|---|---|
| js | function **set_range**( **newval**) |
| nodejs | function **set_range**( **newval**) |
| php | function **set_range**( **$newval**) |
| cpp | int **set_range**( int **newval**) |
| m | -(int) setRange : (int) **newval** |
| pas | function **set_range**( **newval**: LongInt): integer |
| vb | function **set_range**( ByVal **newval** As Integer) As Integer |
| cs | int **set_range**( int **newval**) |
| java | int **set_range**( int **newval**) |
| py | def **set_range**( **newval**) |
| cmd | YServo **target** **set_range** **newval** |

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms], When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

**Parameters :**

**newval** an integer corresponding to the range of use of the servo, specified in per cents

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## servo→set_userData() servo→setUserData()[servo setUserData: ] YServo

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## servo→wait_async() YServo

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.38. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_temperature.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YTemperature = yoctolib.YTemperature; |
| php | require_once('yocto_temperature.php'); |
| cpp | #include "yocto_temperature.h" |
| m | #import "yocto_temperature.h" |
| pas | uses yocto_temperature; |
| vb | yocto_temperature.vb |
| cs | yocto_temperature.cs |
| java | import com.yoctopuce.YoctoAPI.YTemperature; |
| py | from yocto_temperature import * |

**Global functions**

**yFindTemperature(func)**

Retrieves a temperature sensor for a given identifier.

**yFirstTemperature()**

Starts the enumeration of temperature sensors currently accessible.

**`YTemperature` methods**

**temperature→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**temperature→describe()**

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**temperature→get_advertisedValue()**

Returns the current value of the temperature sensor (no more than 6 characters).

**temperature→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**temperature→get_currentValue()**

Returns the current value of the temperature.

**temperature→get_errorMessage()**

Returns the error message of the latest error with the temperature sensor.

**temperature→get_errorType()**

Returns the numerical error code of the latest error with the temperature sensor.

**temperature→get_friendlyName()**

Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**temperature→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**temperature→get_functionId()**

Returns the hardware identifier of the temperature sensor, without reference to the module.

**temperature→get_hardwareId()**

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

**temperature→get_highestValue()**
Returns the maximal value observed for the temperature since the device was started.

**temperature→get_logFrequency()**
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**temperature→get_logicalName()**
Returns the logical name of the temperature sensor.

**temperature→get_lowestValue()**
Returns the minimal value observed for the temperature since the device was started.

**temperature→get_module()**
Gets the `YModule` object for the device on which the function is located.

**temperature→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**temperature→get_recordedData(startTime, endTime)**
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**temperature→get_reportFrequency()**
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**temperature→get_resolution()**
Returns the resolution of the measured values.

**temperature→get_sensorType()**
Returns the temperature sensor type.

**temperature→get_unit()**
Returns the measuring unit for the temperature.

**temperature→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**temperature→isOnline()**
Checks if the temperature sensor is currently reachable, without raising any error.

**temperature→isOnline_async(callback, context)**
Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

**temperature→load(msValidity)**
Preloads the temperature sensor cache with a specified validity duration.

**temperature→loadCalibrationPoints(rawValues, refValues)**
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**temperature→load_async(msValidity, callback, context)**
Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

**temperature→nextTemperature()**
Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

**temperature→registerTimedReportCallback(callback)**
Registers the callback function that is invoked on every periodic timed notification.

**temperature→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**temperature→set_highestValue(newval)**
Changes the recorded maximal value observed.

**temperature→set_logFrequency(newval)**
Changes the datalogger recording frequency for this function.

**temperature→set_logicalName(newval)**

Changes the logical name of the temperature sensor.

**temperature→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**temperature→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**temperature→set_resolution(newval)**

Changes the resolution of the measured physical values.

**temperature→set_sensorType(newval)**

Modify the temperature sensor type.

**temperature→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**temperature→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YTemperature.FindTemperature() yFindTemperature()yFindTemperature() YTemperature

Retrieves a temperature sensor for a given identifier.

| | |
|---|---|
| js | function **yFindTemperature**( **func**) |
| nodejs | function **FindTemperature**( **func**) |
| php | function **yFindTemperature**( **$func**) |
| cpp | YTemperature* **yFindTemperature**( const string& **func**) |
| m | YTemperature* **yFindTemperature**( NSString* **func**) |
| pas | function **yFindTemperature**( **func**: string): TYTemperature |
| vb | function **yFindTemperature**( ByVal **func** As String) As YTemperature |
| cs | YTemperature **FindTemperature**( string **func**) |
| java | YTemperature **FindTemperature**( String **func**) |
| py | def **FindTemperature**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the temperature sensor

**Returns :**

a `YTemperature` object allowing you to drive the temperature sensor.

## YTemperature.FirstTemperature() yFirstTemperature()yFirstTemperature() YTemperature

Starts the enumeration of temperature sensors currently accessible.

| | |
|---|---|
| js | function **yFirstTemperature**( ) |
| nodejs | function **FirstTemperature**( ) |
| php | function **yFirstTemperature**( ) |
| cpp | YTemperature* **yFirstTemperature**( ) |
| m | YTemperature* **yFirstTemperature**( ) |
| pas | function **yFirstTemperature**( ): TYTemperature |
| vb | function **yFirstTemperature**( ) As YTemperature |
| cs | YTemperature **FirstTemperature**( ) |
| java | YTemperature **FirstTemperature**( ) |
| py | def **FirstTemperature**( ) |

Use the method `YTemperature.nextTemperature()` to iterate on next temperature sensors.

**Returns :**

a pointer to a `YTemperature` object, corresponding to the first temperature sensor currently online, or a `null` pointer if there are none.

### temperature→calibrateFromPoints()[temperature calibrateFromPoints: ] YTemperature

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YTemperature **target** **calibrateFromPoints** **rawValues** **refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→describe()[temperature describe] YTemperature

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the temperature sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## temperature→get_advertisedValue() temperature→advertisedValue()[temperature advertisedValue] YTemperature

Returns the current value of the temperature sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YTemperature **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the temperature sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## temperature→get_currentRawValue() temperature→currentRawValue()[temperature currentRawValue] YTemperature

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YTemperature **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## temperature→get_currentValue() temperature→currentValue()[temperature currentValue] YTemperature

Returns the current value of the temperature.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YTemperature **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current value of the temperature

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## temperature→get_errorMessage() temperature→errorMessage()[temperature errorMessage] YTemperature

Returns the error message of the latest error with the temperature sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the temperature sensor object

## temperature→get_errorType() temperature→errorType() YTemperature

Returns the numerical error code of the latest error with the temperature sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the temperature sensor object

## temperature→get_friendlyName() temperature→friendlyName()[temperature friendlyName] YTemperature

Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the temperature sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the temperature sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the temperature sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## temperature→get_functionDescriptor() temperature→functionDescriptor()[temperature functionDescriptor] YTemperature

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## temperature→get_functionId() temperature→functionId()[temperature functionId] YTemperature

Returns the hardware identifier of the temperature sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the temperature sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

### temperature→get_hardwareId() temperature→hardwareId()[temperature hardwareId] YTemperature

Returns the unique hardware identifier of the temperature sensor in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the temperature sensor. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the temperature sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## temperature→get_highestValue() temperature→highestValue()[temperature highestValue] YTemperature

Returns the maximal value observed for the temperature since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YTemperature **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## temperature→get_logFrequency() temperature→logFrequency()[temperature logFrequency] YTemperature

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YTemperature **target** **get_logFrequency** |

**Returns :**
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## temperature→get_logicalName() temperature→logicalName()[temperature logicalName] YTemperature

Returns the logical name of the temperature sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YTemperature **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the temperature sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## temperature→get_lowestValue() temperature→lowestValue()[temperature lowestValue] YTemperature

Returns the minimal value observed for the temperature since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YTemperature **target** **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 module]

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## temperature→get_module_async() temperature→module_async() YTemperature

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## temperature→get_recordedData() temperature→recordedData()[temperature recordedData: ] YTemperature

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YTemperature **target get_recordedData startTime endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## temperature→get_reportFrequency() temperature→reportFrequency()[temperature reportFrequency] YTemperature

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YTemperature **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## temperature→get_resolution() temperature→resolution()[temperature resolution] YTemperature

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YTemperature **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## temperature→get_sensorType() temperature→sensorType()[temperature sensorType] YTemperature

Returns the temperature sensor type.

| | |
|---|---|
| js | function **get_sensorType**( ) |
| nodejs | function **get_sensorType**( ) |
| php | function **get_sensorType**( ) |
| cpp | Y_SENSORTYPE_enum **get_sensorType**( ) |
| m | -(Y_SENSORTYPE_enum) sensorType |
| pas | function **get_sensorType**( ): Integer |
| vb | function **get_sensorType**( ) As Integer |
| cs | int **get_sensorType**( ) |
| java | int **get_sensorType**( ) |
| py | def **get_sensorType**( ) |
| cmd | YTemperature **target** **get_sensorType** |

**Returns :**

a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES corresponding to the temperature sensor type

On failure, throws an exception or returns Y_SENSORTYPE_INVALID.

## temperature→get_unit() temperature→unit()[temperature unit]

**YTemperature**

Returns the measuring unit for the temperature.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YTemperature **target 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() temperature→userData()[temperature userData] YTemperature

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## temperature→isOnline()[temperature isOnline] YTemperature

Checks if the temperature sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

**Returns :**

`true` if the temperature sensor can be reached, and `false` otherwise

## temperature→isOnline_async() YTemperature

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## temperature→load()[temperature load: ] YTemperature

Preloads the temperature sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## temperature→loadCalibrationPoints()[temperature loadCalibrationPoints: ] YTemperature

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YTemperature **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→load_async() YTemperature

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## temperature→nextTemperature()[temperature nextTemperature] YTemperature

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

| | |
|---|---|
| js | function **nextTemperature**( ) |
| nodejs | function **nextTemperature**( ) |
| php | function **nextTemperature**( ) |
| cpp | YTemperature * **nextTemperature**( ) |
| m | -(YTemperature*) **nextTemperature** |
| pas | function **nextTemperature**( ): TYTemperature |
| vb | function **nextTemperature**( ) As YTemperature |
| cs | YTemperature **nextTemperature**( ) |
| java | YTemperature **nextTemperature**( ) |
| py | def **nextTemperature**( ) |

**Returns :**

a pointer to a `YTemperature` object, corresponding to a temperature sensor currently online, or a `null` pointer if there are no more temperature sensors to enumerate.

## temperature→registerTimedReportCallback() [temperature registerTimedReportCallback: ] YTemperature

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YTemperatureTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YTemperatureTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYTemperatureTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## temperature→registerValueCallback()[temperature registerValueCallback: ] YTemperature

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YTemperatureValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YTemperatureValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYTemperatureValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## temperature→set_highestValue() temperature→setHighestValue()[temperature setHighestValue: ] YTemperature

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YTemperature **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→set_logFrequency() temperature→setLogFrequency()[temperature setLogFrequency: ] YTemperature

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YTemperature **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→set_logicalName() temperature→setLogicalName()[temperature setLogicalName: ] YTemperature

Changes the logical name of the temperature sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YTemperature **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the temperature sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## temperature→set_lowestValue() temperature→setLowestValue()[temperature setLowestValue: ] YTemperature

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YTemperature **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→set_reportFrequency() temperature→setReportFrequency()[temperature setReportFrequency: ] YTemperature

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YTemperature **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→set_resolution() temperature→setResolution()[temperature setResolution: ] YTemperature

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YTemperature **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→set_sensorType() temperature→setSensorType()[temperature setSensorType: ] YTemperature

Modify the temperature sensor type.

| | |
|---|---|
| js | function **set_sensorType**( **newval**) |
| nodejs | function **set_sensorType**( **newval**) |
| php | function **set_sensorType**( **$newval**) |
| cpp | int **set_sensorType**( Y_SENSORTYPE_enum **newval**) |
| m | -(int) setSensorType : (Y_SENSORTYPE_enum) **newval** |
| pas | function **set_sensorType**( **newval**: Integer): integer |
| vb | function **set_sensorType**( ByVal **newval** As Integer) As Integer |
| cs | int **set_sensorType**( int **newval**) |
| java | int **set_sensorType**( int **newval**) |
| py | def **set_sensorType**( **newval**) |
| cmd | YTemperature **target set_sensorType newval** |

This function is used to to define the type of thermocouple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a value among `Y_SENSORTYPE_DIGITAL`, `Y_SENSORTYPE_TYPE_K`, `Y_SENSORTYPE_TYPE_E`, `Y_SENSORTYPE_TYPE_J`, `Y_SENSORTYPE_TYPE_N`, `Y_SENSORTYPE_TYPE_R`, `Y_SENSORTYPE_TYPE_S`, `Y_SENSORTYPE_TYPE_T`, `Y_SENSORTYPE_PT100_4WIRES`, `Y_SENSORTYPE_PT100_3WIRES` and `Y_SENSORTYPE_PT100_2WIRES`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## temperature→set_userData() temperature→setUserData()[temperature setUserData: ] YTemperature

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## temperature→wait_async() YTemperature

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.39. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_tilt.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YTilt = yoctolib.YTilt; |
| php | require_once('yocto_tilt.php'); |
| cpp | #include "yocto_tilt.h" |
| m | #import "yocto_tilt.h" |
| pas | uses yocto_tilt; |
| vb | yocto_tilt.vb |
| cs | yocto_tilt.cs |
| java | import com.yoctopuce.YoctoAPI.YTilt; |
| py | from yocto_tilt import * |

**Global functions**

**yFindTilt(func)**

Retrieves a tilt sensor for a given identifier.

**yFirstTilt()**

Starts the enumeration of tilt sensors currently accessible.

**`YTilt` methods**

**tilt→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**tilt→describe()**

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**tilt→get_advertisedValue()**

Returns the current value of the tilt sensor (no more than 6 characters).

**tilt→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**tilt→get_currentValue()**

Returns the current value of the inclination.

**tilt→get_errorMessage()**

Returns the error message of the latest error with the tilt sensor.

**tilt→get_errorType()**

Returns the numerical error code of the latest error with the tilt sensor.

**tilt→get_friendlyName()**

Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**tilt→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**tilt→get_functionId()**

Returns the hardware identifier of the tilt sensor, without reference to the module.

**tilt→get_hardwareId()**

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

**tilt→get_highestValue()**

Returns the maximal value observed for the inclination since the device was started.

**tilt→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**tilt→get_logicalName()**

Returns the logical name of the tilt sensor.

**tilt→get_lowestValue()**

Returns the minimal value observed for the inclination since the device was started.

**tilt→get_module()**

Gets the `YModule` object for the device on which the function is located.

**tilt→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**tilt→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**tilt→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**tilt→get_resolution()**

Returns the resolution of the measured values.

**tilt→get_unit()**

Returns the measuring unit for the inclination.

**tilt→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**tilt→isOnline()**

Checks if the tilt sensor is currently reachable, without raising any error.

**tilt→isOnline_async(callback, context)**

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

**tilt→load(msValidity)**

Preloads the tilt sensor cache with a specified validity duration.

**tilt→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**tilt→load_async(msValidity, callback, context)**

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

**tilt→nextTilt()**

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

**tilt→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**tilt→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**tilt→set_highestValue(newval)**

Changes the recorded maximal value observed.

**tilt→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**tilt→set_logicalName(newval)**

Changes the logical name of the tilt sensor.

**tilt→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**tilt→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**tilt→set_resolution(newval)**

Changes the resolution of the measured physical values.

**tilt→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**tilt→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YTilt.FindTilt() yFindTilt()yFindTilt() YTilt

Retrieves a tilt sensor for a given identifier.

| | |
|---|---|
| js | function **yFindTilt**( **func**) |
| nodejs | function **FindTilt**( **func**) |
| php | function **yFindTilt**( **$func**) |
| cpp | YTilt* **yFindTilt**( const string& **func**) |
| m | YTilt* **yFindTilt**( NSString* **func**) |
| pas | function **yFindTilt**( **func**: string): TYTilt |
| vb | function **yFindTilt**( ByVal **func** As String) As YTilt |
| cs | YTilt **FindTilt**( string **func**) |
| java | YTilt **FindTilt**( String **func**) |
| py | def **FindTilt**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the tilt sensor

**Returns :**

a `YTilt` object allowing you to drive the tilt sensor.

## YTilt.FirstTilt() yFirstTilt()yFirstTilt()

**YTilt**

Starts the enumeration of tilt sensors currently accessible.

| | |
|---|---|
| js | function **yFirstTilt**( ) |
| nodejs | function **FirstTilt**( ) |
| php | function **yFirstTilt**( ) |
| cpp | YTilt* **yFirstTilt**( ) |
| m | YTilt* **yFirstTilt**( ) |
| pas | function **yFirstTilt**( ): TYTilt |
| vb | function **yFirstTilt**( ) As YTilt |
| cs | YTilt **FirstTilt**( ) |
| java | YTilt **FirstTilt**( ) |
| py | def **FirstTilt**( ) |

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

**Returns :**

a pointer to a `YTilt` object, corresponding to the first tilt sensor currently online, or a `null` pointer if there are none.

## tilt→calibrateFromPoints()[tilt calibrateFromPoints: ] YTilt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**,<br>vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues**<br>: (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray,<br>**refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**,<br>List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**,<br>ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YTilt **target calibrateFromPoints rawValues refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→describe()[tilt describe] YTilt

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the tilt sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## tilt→get_advertisedValue() tilt→advertisedValue()[tilt advertisedValue] YTilt

Returns the current value of the tilt sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YTilt **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the tilt sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## tilt→get_currentRawValue()<br>tilt→currentRawValue()[tilt currentRawValue] YTilt

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YTilt **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## tilt→get_currentValue() tilt→currentValue()[tilt currentValue] YTilt

Returns the current value of the inclination.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YTilt **target** **get_currentValue** |

**Returns :**
a floating point number corresponding to the current value of the inclination

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## tilt→get_errorMessage() tilt→errorMessage()[tilt errorMessage] YTilt

Returns the error message of the latest error with the tilt sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the tilt sensor object

## tilt→get_errorType() tilt→errorType() YTilt

Returns the numerical error code of the latest error with the tilt sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the tilt sensor object

## tilt→get_friendlyName() tilt→friendlyName()[tilt friendlyName] YTilt

Returns a global identifier of the tilt sensor in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the tilt sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the tilt sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the tilt sensor using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## tilt→get_functionDescriptor() tilt→functionDescriptor()[tilt functionDescriptor] YTilt

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## tilt→get_functionId() tilt→functionId()[tilt functionId]

YTilt

Returns the hardware identifier of the tilt sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the tilt sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## tilt→get_hardwareId() tilt→hardwareId()[tilt hardwareId] YTilt

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the tilt sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the tilt sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## tilt→get_highestValue() tilt→highestValue()[tilt highestValue] YTilt

Returns the maximal value observed for the inclination since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YTilt **target** **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 logFrequency]

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YTilt **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## tilt→get_logicalName() tilt→logicalName()[tilt logicalName] YTilt

Returns the logical name of the tilt sensor.

| | |
|---|---|
| `js` | function **get_logicalName**( ) |
| `nodejs` | function **get_logicalName**( ) |
| `php` | function **get_logicalName**( ) |
| `cpp` | string **get_logicalName**( ) |
| `m` | -(NSString*) logicalName |
| `pas` | function **get_logicalName**( ): string |
| `vb` | function **get_logicalName**( ) As String |
| `cs` | string **get_logicalName**( ) |
| `java` | String **get_logicalName**( ) |
| `py` | def **get_logicalName**( ) |
| `cmd` | YTilt **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the tilt sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## tilt→get_lowestValue() tilt→lowestValue()[tilt lowestValue] YTilt

Returns the minimal value observed for the inclination since the device was started.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YTilt **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## tilt→get_module() tilt→module()[tilt module] YTilt

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## tilt→get_module_async()
## tilt→module_async()

**YTilt**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## tilt→get_recordedData() tilt→recordedData()[tilt recordedData: ] YTilt

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YTilt **target** **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 reportFrequency]

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YTilt **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## tilt→get_resolution() tilt→resolution()[tilt resolution] YTilt

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YTilt **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## tilt→get_unit() tilt→unit()[tilt unit] YTilt

Returns the measuring unit for the inclination.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YTilt **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## tilt→get_userData()
## tilt→userData()[tilt userData]

YTilt

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## tilt→isOnline()[tilt isOnline] YTilt

Checks if the tilt sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

**Returns :**

`true` if the tilt sensor can be reached, and `false` otherwise

## tilt→isOnline_async() YTilt

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## tilt→load()[tilt load: ] YTilt

Preloads the tilt sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## tilt→loadCalibrationPoints()[tilt loadCalibrationPoints: ] YTilt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YTilt **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→load_async() YTilt

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## tilt→nextTilt()[tilt nextTilt] **YTilt**

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

| | |
|---|---|
| `js` | function **nextTilt**( ) |
| `nodejs` | function **nextTilt**( ) |
| `php` | function **nextTilt**( ) |
| `cpp` | YTilt * **nextTilt**( ) |
| `m` | -(YTilt*) **nextTilt** |
| `pas` | function **nextTilt**( ): TYTilt |
| `vb` | function **nextTilt**( ) As YTilt |
| `cs` | YTilt **nextTilt**( ) |
| `java` | YTilt **nextTilt**( ) |
| `py` | def **nextTilt**( ) |

**Returns :**

a pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a `null` pointer if there are no more tilt sensors to enumerate.

## tilt→registerTimedReportCallback()[tilt registerTimedReportCallback: ] YTilt

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YTiltTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YTiltTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYTiltTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## tilt→registerValueCallback()[tilt registerValueCallback: ] YTilt

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YTiltValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YTiltValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYTiltValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## tilt→set_highestValue() tilt→setHighestValue()[tilt setHighestValue: ] YTilt

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YTilt **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→set_logFrequency() tilt→setLogFrequency()[tilt setLogFrequency: ] YTilt

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YTilt **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→set_logicalName() tilt→setLogicalName()[tilt setLogicalName: ] YTilt

Changes the logical name of the tilt sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YTilt **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the tilt sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## tilt→set_lowestValue() tilt→setLowestValue()[tilt setLowestValue: ] YTilt

Changes the recorded minimal value observed.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YTilt **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→set_reportFrequency() tilt→setReportFrequency()[tilt setReportFrequency: ] YTilt

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YTilt **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→set_resolution() tilt→setResolution()[tilt setResolution: ] YTilt

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YTilt **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## tilt→set_userData()
## tilt→setUserData()[tilt setUserData: ] YTilt

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## tilt→wait_async() YTilt

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.40. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_voc.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YVoc = yoctolib.YVoc; |
| php | require_once('yocto_voc.php'); |
| cpp | #include "yocto_voc.h" |
| m | #import "yocto_voc.h" |
| pas | uses yocto_voc; |
| vb | yocto_voc.vb |
| cs | yocto_voc.cs |
| java | import com.yoctopuce.YoctoAPI.YVoc; |
| py | from yocto_voc import * |

**Global functions**

**yFindVoc(func)**
Retrieves a Volatile Organic Compound sensor for a given identifier.

**yFirstVoc()**
Starts the enumeration of Volatile Organic Compound sensors currently accessible.

**YVoc methods**

**voc→calibrateFromPoints(rawValues, refValues)**
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**voc→describe()**
Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**voc→get_advertisedValue()**
Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

**voc→get_currentRawValue()**
Returns the unrounded and uncalibrated raw value returned by the sensor.

**voc→get_currentValue()**
Returns the current measure for the estimated VOC concentration.

**voc→get_errorMessage()**
Returns the error message of the latest error with the Volatile Organic Compound sensor.

**voc→get_errorType()**
Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

**voc→get_friendlyName()**
Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**voc→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**voc→get_functionId()**
Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

**voc→get_hardwareId()**

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

**voc→get_highestValue()**

Returns the maximal value observed for the estimated VOC concentration.

**voc→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**voc→get_logicalName()**

Returns the logical name of the Volatile Organic Compound sensor.

**voc→get_lowestValue()**

Returns the minimal value observed for the estimated VOC concentration.

**voc→get_module()**

Gets the `YModule` object for the device on which the function is located.

**voc→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**voc→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**voc→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**voc→get_resolution()**

Returns the resolution of the measured values.

**voc→get_unit()**

Returns the measuring unit for the estimated VOC concentration.

**voc→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**voc→isOnline()**

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

**voc→isOnline_async(callback, context)**

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

**voc→load(msValidity)**

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

**voc→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**voc→load_async(msValidity, callback, context)**

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

**voc→nextVoc()**

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

**voc→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**voc→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**voc→set_highestValue(newval)**

Changes the recorded maximal value observed for the estimated VOC concentration.

**voc→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**voc→set_logicalName(newval)**

Changes the logical name of the Volatile Organic Compound sensor.

**voc→set_lowestValue(newval)**

Changes the recorded minimal value observed for the estimated VOC concentration.

**voc→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**voc→set_resolution(newval)**

Changes the resolution of the measured physical values.

**voc→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**voc→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YVoc.FindVoc() yFindVoc()yFindVoc() YVoc

Retrieves a Volatile Organic Compound sensor for a given identifier.

| | |
|---|---|
| js | function **yFindVoc**( **func**) |
| nodejs | function **FindVoc**( **func**) |
| php | function **yFindVoc**( **$func**) |
| cpp | YVoc* **yFindVoc**( const string& **func**) |
| m | YVoc* **yFindVoc**( NSString* **func**) |
| pas | function **yFindVoc**( **func**: string): TYVoc |
| vb | function **yFindVoc**( ByVal **func** As String) As YVoc |
| cs | YVoc **FindVoc**( string **func**) |
| java | YVoc **FindVoc**( String **func**) |
| py | def **FindVoc**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Volatile Organic Compound sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoc.isOnline()` to test if the Volatile Organic Compound sensor is indeed online at a given time. In case of ambiguity when looking for a Volatile Organic Compound sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the Volatile Organic Compound sensor

**Returns :**

a `YVoc` object allowing you to drive the Volatile Organic Compound sensor.

## YVoc.FirstVoc() yFirstVoc()yFirstVoc() YVoc

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

| | |
|---|---|
| js | function **yFirstVoc**( ) |
| nodejs | function **FirstVoc**( ) |
| php | function **yFirstVoc**( ) |
| cpp | YVoc* **yFirstVoc**( ) |
| m | YVoc* **yFirstVoc**( ) |
| pas | function **yFirstVoc**( ): TYVoc |
| vb | function **yFirstVoc**( ) As YVoc |
| cs | YVoc **FirstVoc**( ) |
| java | YVoc **FirstVoc**( ) |
| py | def **FirstVoc**( ) |

Use the method `YVoc.nextVoc()` to iterate on next Volatile Organic Compound sensors.

**Returns :**

a pointer to a `YVoc` object, corresponding to the first Volatile Organic Compound sensor currently online, or a `null` pointer if there are none.

## voc→calibrateFromPoints()[voc calibrateFromPoints: ] YVoc

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

| | |
|---|---|
| js | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| nodejs | function **calibrateFromPoints**( **rawValues**, **refValues**) |
| php | function **calibrateFromPoints**( **$rawValues**, **$refValues**) |
| cpp | int **calibrateFromPoints**( vector<double> **rawValues**, vector<double> **refValues**) |
| m | -(int) **calibrateFromPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **calibrateFromPoints**( **rawValues**: TDoubleArray, **refValues**: TDoubleArray): LongInt |
| vb | procedure **calibrateFromPoints**( ) |
| cs | int **calibrateFromPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **calibrateFromPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **calibrateFromPoints**( **rawValues**, **refValues**) |
| cmd | YVoc **target calibrateFromPoints rawValues refValues** |

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→describe()[voc describe] YVoc

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the Volatile Organic Compound sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## voc→get_advertisedValue() voc→advertisedValue()[voc advertisedValue] YVoc

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YVoc **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## voc→get_currentRawValue() voc→currentRawValue()[voc currentRawValue] YVoc

Returns the unrounded and uncalibrated raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YVoc **target** **get_currentRawValue** |

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## voc→get_currentValue() voc→currentValue()[voc currentValue] YVoc

Returns the current measure for the estimated VOC concentration.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YVoc **target** **get_currentValue** |

**Returns :**

a floating point number corresponding to the current measure for the estimated VOC concentration

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## voc→get_errorMessage() voc→errorMessage()[voc errorMessage] YVoc

Returns the error message of the latest error with the Volatile Organic Compound sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the Volatile Organic Compound sensor object

## voc→get_errorType() voc→errorType() YVoc

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the Volatile Organic Compound sensor object

## voc→get_friendlyName() voc→friendlyName()[voc friendlyName] YVoc

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the Volatile Organic Compound sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the Volatile Organic Compound sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## voc→get_functionDescriptor() voc→functionDescriptor()[voc functionDescriptor] YVoc

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## voc→get_functionId() voc→functionId()[voc functionId] YVoc

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the Volatile Organic Compound sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## voc→get_hardwareId() voc→hardwareId()[voc hardwareId] YVoc

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Volatile Organic Compound sensor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the Volatile Organic Compound sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## voc→get_highestValue() voc→highestValue()[voc highestValue] YVoc

Returns the maximal value observed for the estimated VOC concentration.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YVoc **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the estimated VOC concentration

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## voc→get_logFrequency() voc→logFrequency()[voc logFrequency] YVoc

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YVoc **target** **get_logFrequency** |

**Returns :**
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## voc→get_logicalName() voc→logicalName()[voc logicalName] YVoc

Returns the logical name of the Volatile Organic Compound sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YVoc **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the Volatile Organic Compound sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## voc→get_lowestValue() voc→lowestValue()[voc lowestValue] YVoc

Returns the minimal value observed for the estimated VOC concentration.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YVoc **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the estimated VOC concentration

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## voc→get_module() voc→module()[voc module] — YVoc

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## voc→get_module_async() voc→module_async() YVoc

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## voc→get_recordedData() voc→recordedData()[voc recordedData: ] YVoc

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YVoc **target** **get_recordedData** **startTime** **endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## voc→get_reportFrequency() voc→reportFrequency()[voc reportFrequency] YVoc

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YVoc **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## voc→get_resolution() voc→resolution()[voc resolution] YVoc

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| nodejs | function **get_resolution**( ) |
| php | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | function **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| py | def **get_resolution**( ) |
| cmd | YVoc **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## voc→get_unit()
## voc→unit()[voc unit]

**YVoc**

Returns the measuring unit for the estimated VOC concentration.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YVoc **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## voc→get_userData() voc→userData()[voc userData] — YVoc

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## voc→isOnline()[voc isOnline] YVoc

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Volatile Organic Compound sensor.

**Returns :**

`true` if the Volatile Organic Compound sensor can be reached, and `false` otherwise

## voc→isOnline_async() YVoc

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

- **callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
- **context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## voc→load()[voc load: ] YVoc

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## voc→loadCalibrationPoints()[voc loadCalibrationPoints: ] YVoc

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YVoc **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→load_async() YVoc

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

`js` function **load_async**( **msValidity**, **callback**, **context**)

`nodejs` function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## voc→nextVoc()[voc nextVoc] YVoc

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

| | |
|---|---|
| js | function **nextVoc**( ) |
| nodejs | function **nextVoc**( ) |
| php | function **nextVoc**( ) |
| cpp | YVoc * **nextVoc**( ) |
| m | -(YVoc*) **nextVoc** |
| pas | function **nextVoc**( ): TYVoc |
| vb | function **nextVoc**( ) As YVoc |
| cs | YVoc **nextVoc**( ) |
| java | YVoc **nextVoc**( ) |
| py | def **nextVoc**( ) |

**Returns :**

a pointer to a `YVoc` object, corresponding to a Volatile Organic Compound sensor currently online, or a `null` pointer if there are no more Volatile Organic Compound sensors to enumerate.

## voc→registerTimedReportCallback()[voc registerTimedReportCallback: ] YVoc

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YVocTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YVocTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYVocTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## voc→registerValueCallback()[voc registerValueCallback: ] YVoc

Registers the callback function that is invoked on every change of advertised value.

```
js      function registerValueCallback( callback)
nodejs  function registerValueCallback( callback)
php     function registerValueCallback( $callback)
cpp     int registerValueCallback( YVocValueCallback callback)
m       -(int) registerValueCallback : (YVocValueCallback) callback
pas     function registerValueCallback( callback: TYVocValueCallback): LongInt
vb      function registerValueCallback( ) As Integer
cs      int registerValueCallback( ValueCallback callback)
java    int registerValueCallback( UpdateCallback callback)
py      def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## voc→set_highestValue() voc→setHighestValue()[voc setHighestValue: ] YVoc

Changes the recorded maximal value observed for the estimated VOC concentration.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YVoc **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the estimated VOC concentration

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→set_logFrequency() voc→setLogFrequency()[voc setLogFrequency: ] YVoc

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YVoc **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→set_logicalName() voc→setLogicalName()[voc setLogicalName: ] YVoc

Changes the logical name of the Volatile Organic Compound sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YVoc **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the Volatile Organic Compound sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## voc→set_lowestValue() voc→setLowestValue()[voc setLowestValue: ] YVoc

Changes the recorded minimal value observed for the estimated VOC concentration.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YVoc **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the estimated VOC concentration

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→set_reportFrequency() voc→setReportFrequency()[voc setReportFrequency: ] YVoc

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YVoc **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→set_resolution() voc→setResolution()[voc setResolution: ] YVoc

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YVoc **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→set_userData() voc→setUserData()[voc setUserData: ] YVoc

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## voc→wait_async() YVoc

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

## 3.41. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_voltage.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YVoltage = yoctolib.YVoltage; |
| php | require_once('yocto_voltage.php'); |
| cpp | #include "yocto_voltage.h" |
| m | #import "yocto_voltage.h" |
| pas | uses yocto_voltage; |
| vb | yocto_voltage.vb |
| cs | yocto_voltage.cs |
| java | import com.yoctopuce.YoctoAPI.YVoltage; |
| py | from yocto_voltage import * |

**Global functions**

**yFindVoltage(func)**

Retrieves a voltage sensor for a given identifier.

**yFirstVoltage()**

Starts the enumeration of voltage sensors currently accessible.

**`YVoltage` methods**

**voltage→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**voltage→describe()**

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**voltage→get_advertisedValue()**

Returns the current value of the voltage sensor (no more than 6 characters).

**voltage→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

**voltage→get_currentValue()**

Returns the current measure for the voltage.

**voltage→get_errorMessage()**

Returns the error message of the latest error with the voltage sensor.

**voltage→get_errorType()**

Returns the numerical error code of the latest error with the voltage sensor.

**voltage→get_friendlyName()**

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**voltage→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**voltage→get_functionId()**

Returns the hardware identifier of the voltage sensor, without reference to the module.

**voltage→get_hardwareId()**

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

**voltage→get_highestValue()**

Returns the maximal value observed for the voltage.

**voltage→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**voltage→get_logicalName()**

Returns the logical name of the voltage sensor.

**voltage→get_lowestValue()**

Returns the minimal value observed for the voltage.

**voltage→get_module()**

Gets the `YModule` object for the device on which the function is located.

**voltage→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**voltage→get_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**voltage→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**voltage→get_resolution()**

Returns the resolution of the measured values.

**voltage→get_unit()**

Returns the measuring unit for the voltage.

**voltage→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**voltage→isOnline()**

Checks if the voltage sensor is currently reachable, without raising any error.

**voltage→isOnline_async(callback, context)**

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

**voltage→load(msValidity)**

Preloads the voltage sensor cache with a specified validity duration.

**voltage→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**voltage→load_async(msValidity, callback, context)**

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

**voltage→nextVoltage()**

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

**voltage→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**voltage→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**voltage→set_highestValue(newval)**

Changes the recorded maximal value observed pour the voltage.

**voltage→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**voltage→set_logicalName(newval)**

Changes the logical name of the voltage sensor.

**voltage→set_lowestValue(newval)**

Changes the recorded minimal value observed pour the voltage.

**voltage→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**voltage→set_resolution(newval)**

Changes the resolution of the measured values.

**voltage→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**voltage→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YVoltage.FindVoltage() yFindVoltage()yFindVoltage() YVoltage

Retrieves a voltage sensor for a given identifier.

| | |
|---|---|
| js | function **yFindVoltage**( **func**) |
| nodejs | function **FindVoltage**( **func**) |
| php | function **yFindVoltage**( **$func**) |
| cpp | YVoltage* **yFindVoltage**( const string& **func**) |
| m | YVoltage* **yFindVoltage**( NSString* **func**) |
| pas | function **yFindVoltage**( **func**: string): TYVoltage |
| vb | function **yFindVoltage**( ByVal **func** As String) As YVoltage |
| cs | YVoltage **FindVoltage**( string **func**) |
| java | YVoltage **FindVoltage**( String **func**) |
| py | def **FindVoltage**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the voltage sensor

**Returns :**

a `YVoltage` object allowing you to drive the voltage sensor.

## YVoltage.FirstVoltage() yFirstVoltage()yFirstVoltage()

YVoltage

Starts the enumeration of voltage sensors currently accessible.

| | |
|---|---|
| js | function **yFirstVoltage**( ) |
| nodejs | function **FirstVoltage**( ) |
| php | function **yFirstVoltage**( ) |
| cpp | YVoltage* **yFirstVoltage**( ) |
| m | YVoltage* **yFirstVoltage**( ) |
| pas | function **yFirstVoltage**( ): TYVoltage |
| vb | function **yFirstVoltage**( ) As YVoltage |
| cs | YVoltage **FirstVoltage**( ) |
| java | YVoltage **FirstVoltage**( ) |
| py | def **FirstVoltage**( ) |

Use the method `YVoltage.nextVoltage()` to iterate on next voltage sensors.

**Returns :**

a pointer to a `YVoltage` object, corresponding to the first voltage sensor currently online, or a `null` pointer if there are none.

## voltage→calibrateFromPoints()[voltage calibrateFromPoints: ] YVoltage

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
js      function calibrateFromPoints( rawValues, refValues)
nodejs  function calibrateFromPoints( rawValues, refValues)
php     function calibrateFromPoints( $rawValues, $refValues)
cpp     int calibrateFromPoints( vector<double> rawValues,
                                 vector<double> refValues)
m       -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                   : (NSMutableArray*) refValues
pas     function calibrateFromPoints( rawValues: TDoubleArray,
                                      refValues: TDoubleArray): LongInt
vb      procedure calibrateFromPoints( )
cs      int calibrateFromPoints( List<double> rawValues,
                                 List<double> refValues)
java    int calibrateFromPoints( ArrayList<Double> rawValues,
                                 ArrayList<Double> refValues)
py      def calibrateFromPoints( rawValues, refValues)
cmd     YVoltage target calibrateFromPoints rawValues refValues
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→describe()[voltage describe] YVoltage

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the voltage sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## voltage→get_advertisedValue() voltage→advertisedValue()[voltage advertisedValue] YVoltage

Returns the current value of the voltage sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YVoltage **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the voltage sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## voltage→get_currentRawValue() voltage→currentRawValue()[voltage currentRawValue] YVoltage

Returns the uncalibrated, unrounded raw value returned by the sensor.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| nodejs | function **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | function **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| py | def **get_currentRawValue**( ) |
| cmd | YVoltage **target** **get_currentRawValue** |

**Returns :**
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## voltage→get_currentValue() voltage→currentValue()[voltage currentValue] YVoltage

Returns the current measure for the voltage.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| nodejs | function **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | function **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| py | def **get_currentValue**( ) |
| cmd | YVoltage **target** **get_currentValue** |

**Returns :**
a floating point number corresponding to the current measure for the voltage

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## voltage→get_errorMessage() voltage→errorMessage()[voltage errorMessage] YVoltage

Returns the error message of the latest error with the voltage sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the voltage sensor object

## voltage→get_errorType() voltage→errorType() YVoltage

Returns the numerical error code of the latest error with the voltage sensor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the voltage sensor object

## voltage→get_friendlyName() voltage→friendlyName()[voltage friendlyName] YVoltage

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| `js` | function **get_friendlyName**( ) |
| `nodejs` | function **get_friendlyName**( ) |
| `php` | function **get_friendlyName**( ) |
| `cpp` | string **get_friendlyName**( ) |
| `m` | -(NSString*) friendlyName |
| `cs` | string **get_friendlyName**( ) |
| `java` | String **get_friendlyName**( ) |
| `py` | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the voltage sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage sensor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the voltage sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## voltage→get_functionDescriptor() voltage→functionDescriptor()[voltage functionDescriptor] YVoltage

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## voltage→get_functionId() voltage→functionId()[voltage functionId] YVoltage

Returns the hardware identifier of the voltage sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the voltage sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## voltage→get_hardwareId() voltage→hardwareId()[voltage hardwareId] YVoltage

Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the voltage sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## voltage→get_highestValue() voltage→highestValue()[voltage highestValue] YVoltage

Returns the maximal value observed for the voltage.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| nodejs | function **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | function **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| py | def **get_highestValue**( ) |
| cmd | YVoltage **target** **get_highestValue** |

**Returns :**

a floating point number corresponding to the maximal value observed for the voltage

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## voltage→get_logFrequency() voltage→logFrequency()[voltage logFrequency] YVoltage

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| nodejs | function **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | function **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| py | def **get_logFrequency**( ) |
| cmd | YVoltage **target** **get_logFrequency** |

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

## voltage→get_logicalName() voltage→logicalName()[voltage logicalName] YVoltage

Returns the logical name of the voltage sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YVoltage **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the voltage sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## voltage→get_lowestValue() voltage→lowestValue()[voltage lowestValue] YVoltage

Returns the minimal value observed for the voltage.

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| nodejs | function **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | function **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| py | def **get_lowestValue**( ) |
| cmd | YVoltage **target** **get_lowestValue** |

**Returns :**
a floating point number corresponding to the minimal value observed for the voltage

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## voltage→get_module() voltage→module()[voltage module] YVoltage

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## voltage→get_module_async() voltage→module_async() YVoltage

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## voltage→get_recordedData() voltage→recordedData()[voltage recordedData: ] YVoltage

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| | |
|---|---|
| js | function **get_recordedData**( **startTime**, **endTime**) |
| nodejs | function **get_recordedData**( **startTime**, **endTime**) |
| php | function **get_recordedData**( **$startTime**, **$endTime**) |
| cpp | YDataSet **get_recordedData**( s64 **startTime**, s64 **endTime**) |
| m | -(YDataSet*) recordedData : (s64) **startTime**<br>: (s64) **endTime** |
| pas | function **get_recordedData**( **startTime**: int64, **endTime**: int64): TYDataSet |
| vb | function **get_recordedData**( ) As YDataSet |
| cs | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| java | YDataSet **get_recordedData**( long **startTime**, long **endTime**) |
| py | def **get_recordedData**( **startTime**, **endTime**) |
| cmd | YVoltage **target get_recordedData startTime endTime** |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

## voltage→get_reportFrequency() voltage→reportFrequency()[voltage reportFrequency] YVoltage

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| nodejs | function **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | function **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| py | def **get_reportFrequency**( ) |
| cmd | YVoltage **target** **get_reportFrequency** |

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

## voltage→get_resolution() voltage→resolution()[voltage resolution] YVoltage

Returns the resolution of the measured values.

| | |
|---|---|
| `js` | function **get_resolution**( ) |
| `nodejs` | function **get_resolution**( ) |
| `php` | function **get_resolution**( ) |
| `cpp` | double **get_resolution**( ) |
| `m` | -(double) resolution |
| `pas` | function **get_resolution**( ): double |
| `vb` | function **get_resolution**( ) As Double |
| `cs` | double **get_resolution**( ) |
| `java` | double **get_resolution**( ) |
| `py` | def **get_resolution**( ) |
| `cmd` | YVoltage **target** **get_resolution** |

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## voltage→get_unit()
## voltage→unit()[voltage unit]

**YVoltage**

Returns the measuring unit for the voltage.

| | |
|---|---|
| js | function **get_unit**( ) |
| nodejs | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YVoltage **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## voltage→get_userData() voltage→userData()[voltage userData] — YVoltage

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## voltage→isOnline()[voltage isOnline] YVoltage

Checks if the voltage sensor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

**Returns :**
`true` if the voltage sensor can be reached, and `false` otherwise

## voltage→isOnline_async() YVoltage

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## voltage→load()[voltage load: ] YVoltage

Preloads the voltage sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## voltage→loadCalibrationPoints()[voltage loadCalibrationPoints: ] YVoltage

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

| | |
|---|---|
| js | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| nodejs | function **loadCalibrationPoints**( **rawValues**, **refValues**) |
| php | function **loadCalibrationPoints**( **&$rawValues**, **&$refValues**) |
| cpp | int **loadCalibrationPoints**( vector<double>& **rawValues**, vector<double>& **refValues**) |
| m | -(int) **loadCalibrationPoints** : (NSMutableArray*) **rawValues** : (NSMutableArray*) **refValues** |
| pas | function **loadCalibrationPoints**( var **rawValues**: TDoubleArray, var **refValues**: TDoubleArray): LongInt |
| vb | procedure **loadCalibrationPoints**( ) |
| cs | int **loadCalibrationPoints**( List<double> **rawValues**, List<double> **refValues**) |
| java | int **loadCalibrationPoints**( ArrayList<Double> **rawValues**, ArrayList<Double> **refValues**) |
| py | def **loadCalibrationPoints**( **rawValues**, **refValues**) |
| cmd | YVoltage **target** **loadCalibrationPoints** **rawValues** **refValues** |

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→load_async() YVoltage

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

js function **load_async**( **msValidity**, **callback**, **context**)
nodejs function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## voltage→nextVoltage()[voltage nextVoltage] YVoltage

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

| | |
|---|---|
| js | function **nextVoltage**( ) |
| nodejs | function **nextVoltage**( ) |
| php | function **nextVoltage**( ) |
| cpp | YVoltage * **nextVoltage**( ) |
| m | -(YVoltage*) **nextVoltage** |
| pas | function **nextVoltage**( ): TYVoltage |
| vb | function **nextVoltage**( ) As YVoltage |
| cs | YVoltage **nextVoltage**( ) |
| java | YVoltage **nextVoltage**( ) |
| py | def **nextVoltage**( ) |

**Returns :**

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a `null` pointer if there are no more voltage sensors to enumerate.

### voltage→registerTimedReportCallback()[voltage registerTimedReportCallback: ] YVoltage

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| nodejs | function **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| cpp | int **registerTimedReportCallback**( YVoltageTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YVoltageTimedReportCallback) **callback** |
| pas | function **registerTimedReportCallback**( **callback**: TYVoltageTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | def **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

## voltage→registerValueCallback()[voltage registerValueCallback: ] YVoltage

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YVoltageValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YVoltageValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYVoltageValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## voltage→set_highestValue() voltage→setHighestValue()[voltage setHighestValue: ] YVoltage

Changes the recorded maximal value observed pour the voltage.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| nodejs | function **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | function **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| py | def **set_highestValue**( **newval**) |
| cmd | YVoltage **target** **set_highestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed pour the voltage

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→set_logFrequency() voltage→setLogFrequency()[voltage setLogFrequency: ] YVoltage

Changes the datalogger recording frequency for this function.

| | |
|---|---|
| js | function **set_logFrequency**( **newval**) |
| nodejs | function **set_logFrequency**( **newval**) |
| php | function **set_logFrequency**( **$newval**) |
| cpp | int **set_logFrequency**( const string& **newval**) |
| m | -(int) setLogFrequency : (NSString*) **newval** |
| pas | function **set_logFrequency**( **newval**: string): integer |
| vb | function **set_logFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_logFrequency**( string **newval**) |
| java | int **set_logFrequency**( String **newval**) |
| py | def **set_logFrequency**( **newval**) |
| cmd | YVoltage **target** **set_logFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→set_logicalName() voltage→setLogicalName()[voltage setLogicalName: ] YVoltage

Changes the logical name of the voltage sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YVoltage **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the voltage sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## voltage→set_lowestValue() voltage→setLowestValue()[voltage setLowestValue: ] YVoltage

Changes the recorded minimal value observed pour the voltage.

| | |
|---|---|
| js | function **set_lowestValue**( **newval**) |
| nodejs | function **set_lowestValue**( **newval**) |
| php | function **set_lowestValue**( **$newval**) |
| cpp | int **set_lowestValue**( double **newval**) |
| m | -(int) setLowestValue : (double) **newval** |
| pas | function **set_lowestValue**( **newval**: double): integer |
| vb | function **set_lowestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_lowestValue**( double **newval**) |
| java | int **set_lowestValue**( double **newval**) |
| py | def **set_lowestValue**( **newval**) |
| cmd | YVoltage **target** **set_lowestValue** **newval** |

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed pour the voltage

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→set_reportFrequency() voltage→setReportFrequency()[voltage setReportFrequency: ] YVoltage

Changes the timed value notification frequency for this function.

| | |
|---|---|
| js | function **set_reportFrequency**( **newval**) |
| nodejs | function **set_reportFrequency**( **newval**) |
| php | function **set_reportFrequency**( **$newval**) |
| cpp | int **set_reportFrequency**( const string& **newval**) |
| m | -(int) setReportFrequency : (NSString*) **newval** |
| pas | function **set_reportFrequency**( **newval**: string): integer |
| vb | function **set_reportFrequency**( ByVal **newval** As String) As Integer |
| cs | int **set_reportFrequency**( string **newval**) |
| java | int **set_reportFrequency**( String **newval**) |
| py | def **set_reportFrequency**( **newval**) |
| cmd | YVoltage **target** **set_reportFrequency** **newval** |

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→set_resolution() voltage→setResolution()[voltage setResolution: ] YVoltage

Changes the resolution of the measured values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| nodejs | function **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | function **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| py | def **set_resolution**( **newval**) |
| cmd | YVoltage **target** **set_resolution** **newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voltage→set_userData() voltage→setUserData()[voltage setUserData: ] YVoltage

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## voltage→wait_async() YVoltage

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.42. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_vsource.js'></script> |
| php | require_once('yocto_vsource.php'); |
| cpp | #include "yocto_vsource.h" |
| m | #import "yocto_vsource.h" |
| pas | uses yocto_vsource; |
| vb | yocto_vsource.vb |
| cs | yocto_vsource.cs |
| java | import com.yoctopuce.YoctoAPI.YVSource; |
| py | from yocto_vsource import * |

**Global functions**

**yFindVSource(func)**
Retrieves a voltage source for a given identifier.

**yFirstVSource()**
Starts the enumeration of voltage sources currently accessible.

**YVSource methods**

**vsource→describe()**
Returns a short text that describes the function in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**vsource→get_advertisedValue()**
Returns the current value of the voltage source (no more than 6 characters).

**vsource→get_errorMessage()**
Returns the error message of the latest error with this function.

**vsource→get_errorType()**
Returns the numerical error code of the latest error with this function.

**vsource→get_extPowerFailure()**
Returns true if external power supply voltage is too low.

**vsource→get_failure()**
Returns true if the module is in failure mode.

**vsource→get_friendlyName()**
Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

**vsource→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**vsource→get_functionId()**
Returns the hardware identifier of the function, without reference to the module.

**vsource→get_hardwareId()**
Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

**vsource→get_logicalName()**
Returns the logical name of the voltage source.

**vsource→get_module()**
Gets the `YModule` object for the device on which the function is located.

**vsource→get_module_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**vsource→get_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

**vsource→get_overHeat()**

Returns TRUE if the module is overheating.

**vsource→get_overLoad()**

Returns true if the device is not able to maintaint the requested voltage output .

**vsource→get_regulationFailure()**

Returns true if the voltage output is too high regarding the requested voltage .

**vsource→get_unit()**

Returns the measuring unit for the voltage.

**vsource→get_userData()**

Returns the value of the userData attribute, as previously stored using method set_userData.

**vsource→get_voltage()**

Returns the voltage output command (mV)

**vsource→isOnline()**

Checks if the function is currently reachable, without raising any error.

**vsource→isOnline_async(callback, context)**

Checks if the function is currently reachable, without raising any error (asynchronous version).

**vsource→load(msValidity)**

Preloads the function cache with a specified validity duration.

**vsource→load_async(msValidity, callback, context)**

Preloads the function cache with a specified validity duration (asynchronous version).

**vsource→nextVSource()**

Continues the enumeration of voltage sources started using yFirstVSource().

**vsource→pulse(voltage, ms_duration)**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

**vsource→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**vsource→set_logicalName(newval)**

Changes the logical name of the voltage source.

**vsource→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**vsource→set_voltage(newval)**

Tunes the device output voltage (milliVolts).

**vsource→voltageMove(target, ms_duration)**

Performs a smooth move at constant speed toward a given value.

**vsource→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## yFindVSource() — YVSource.FindVSource()yFindVSource() YVSource

Retrieves a voltage source for a given identifier.

| | |
|---|---|
| js | function **yFindVSource**( **func**) |
| php | function **yFindVSource**( **$func**) |
| cpp | YVSource* **yFindVSource**( const string& **func**) |
| m | YVSource* **yFindVSource**( NSString* **func**) |
| pas | function **yFindVSource**( **func**: string): TYVSource |
| vb | function **yFindVSource**( ByVal **func** As String) As YVSource |
| cs | YVSource **FindVSource**( string **func**) |
| java | YVSource **FindVSource**( String **func**) |
| py | def **FindVSource**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the voltage source

**Returns :**

a `YVSource` object allowing you to drive the voltage source.

## yFirstVSource() — YVSource.FirstVSource()yFirstVSource() YVSource

Starts the enumeration of voltage sources currently accessible.

| | |
|---|---|
| js | function **yFirstVSource**( ) |
| php | function **yFirstVSource**( ) |
| cpp | YVSource* **yFirstVSource**( ) |
| m | YVSource* **yFirstVSource**( ) |
| pas | function **yFirstVSource**( ): TYVSource |
| vb | function **yFirstVSource**( ) As YVSource |
| cs | YVSource **FirstVSource**( ) |
| java | YVSource **FirstVSource**( ) |
| py | def **FirstVSource**( ) |

Use the method `YVSource.nextVSource()` to iterate on next voltage sources.

**Returns :**

a pointer to a `YVSource` object, corresponding to the first voltage source currently online, or a `null` pointer if there are none.

## vsource→describe()[vsource describe] YVSource

Returns a short text that describes the function in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the function (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## vsource→get_advertisedValue() vsource→advertisedValue()[vsource advertisedValue] YVSource

Returns the current value of the voltage source (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YVSource **target** **get_advertisedValue** |

**Returns :**
a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## vsource→get_errorMessage() vsource→errorMessage()[vsource errorMessage] YVSource

Returns the error message of the latest error with this function.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using this function object

## vsource→get_errorType() vsource→errorType()

**YVSource**

Returns the numerical error code of the latest error with this function.

| | |
|---|---|
| js | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using this function object

## vsource→get_extPowerFailure() vsource→extPowerFailure()[vsource extPowerFailure] YVSource

Returns true if external power supply voltage is too low.

| | |
|---|---|
| js | function **get_extPowerFailure**( ) |
| php | function **get_extPowerFailure**( ) |
| cpp | Y_EXTPOWERFAILURE_enum **get_extPowerFailure**( ) |
| m | -(Y_EXTPOWERFAILURE_enum) extPowerFailure |
| pas | function **get_extPowerFailure**( ): Integer |
| vb | function **get_extPowerFailure**( ) As Integer |
| cs | int **get_extPowerFailure**( ) |
| java | int **get_extPowerFailure**( ) |
| py | def **get_extPowerFailure**( ) |
| cmd | YVSource **target** **get_extPowerFailure** |

**Returns :**

either `Y_EXTPOWERFAILURE_FALSE` or `Y_EXTPOWERFAILURE_TRUE`, according to true if external power supply voltage is too low

On failure, throws an exception or returns `Y_EXTPOWERFAILURE_INVALID`.

## vsource→get_failure() vsource→failure()[vsource failure] YVSource

Returns true if the module is in failure mode.

| | |
|---|---|
| js | function **get_failure**( ) |
| php | function **get_failure**( ) |
| cpp | Y_FAILURE_enum **get_failure**( ) |
| m | -(Y_FAILURE_enum) failure |
| pas | function **get_failure**( ): Integer |
| vb | function **get_failure**( ) As Integer |
| cs | int **get_failure**( ) |
| java | int **get_failure**( ) |
| py | def **get_failure**( ) |
| cmd | YVSource **target** **get_failure** |

More information can be obtained by testing get_overheat, get_overcurrent etc... When a error condition is met, the output voltage is set to zéro and cannot be changed until the reset() function is called.

**Returns :**

either `Y_FAILURE_FALSE` or `Y_FAILURE_TRUE`, according to true if the module is in failure mode

On failure, throws an exception or returns `Y_FAILURE_INVALID`.

## vsource→get_friendlyName() YVSource
## vsource→friendlyName()[vsource friendlyName]

Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | virtual string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | override string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the function using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## vsource→get_functionDescriptor() vsource→functionDescriptor()[vsource vsourceDescriptor] YVSource

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## vsource→get_functionId() vsource→functionId()[vsource vsourceId] YVSource

Returns the hardware identifier of the function, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the function (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## vsource→get_hardwareId() vsource→hardwareId()[vsource hardwareId] YVSource

Returns the unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the function (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## vsource→get_logicalName() vsource→logicalName()[vsource logicalName] YVSource

Returns the logical name of the voltage source.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YVSource **target** **get_logicalName** |

**Returns :**
a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## vsource→get_module() vsource→module()[vsource module] **YVSource**

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| `js` | function **get_module**( ) |
| `php` | function **get_module**( ) |
| `cpp` | YModule * **get_module**( ) |
| `m` | -(YModule*) module |
| `pas` | function **get_module**( ): TYModule |
| `vb` | function **get_module**( ) As YModule |
| `cs` | YModule **get_module**( ) |
| `java` | YModule **get_module**( ) |
| `py` | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## vsource→get_module_async() vsource→module_async() YVSource

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`js` function **get_module_async**( **callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## vsource→get_overCurrent() vsource→overCurrent()[vsource overCurrent] YVSource

Returns true if the appliance connected to the device is too greedy .

| | |
|---|---|
| js | function **get_overCurrent**( ) |
| php | function **get_overCurrent**( ) |
| cpp | Y_OVERCURRENT_enum **get_overCurrent**( ) |
| m | -(Y_OVERCURRENT_enum) overCurrent |
| pas | function **get_overCurrent**( ): Integer |
| vb | function **get_overCurrent**( ) As Integer |
| cs | int **get_overCurrent**( ) |
| java | int **get_overCurrent**( ) |
| py | def **get_overCurrent**( ) |
| cmd | YVSource **target** **get_overCurrent** |

**Returns :**

either `Y_OVERCURRENT_FALSE` or `Y_OVERCURRENT_TRUE`, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns `Y_OVERCURRENT_INVALID`.

## vsource→get_overHeat() vsource→overHeat()[vsource overHeat] YVSource

Returns TRUE if the module is overheating.

| | |
|---|---|
| js | function **get_overHeat**( ) |
| php | function **get_overHeat**( ) |
| cpp | Y_OVERHEAT_enum **get_overHeat**( ) |
| m | -(Y_OVERHEAT_enum) overHeat |
| pas | function **get_overHeat**( ): Integer |
| vb | function **get_overHeat**( ) As Integer |
| cs | int **get_overHeat**( ) |
| java | int **get_overHeat**( ) |
| py | def **get_overHeat**( ) |
| cmd | YVSource **target** **get_overHeat** |

**Returns :**

either `Y_OVERHEAT_FALSE` or `Y_OVERHEAT_TRUE`, according to TRUE if the module is overheating

On failure, throws an exception or returns `Y_OVERHEAT_INVALID`.

## vsource→get_overLoad() vsource→overLoad()[vsource overLoad] **YVSource**

Returns true if the device is not able to maintaint the requested voltage output .

| | |
|---|---|
| js | function **get_overLoad**( ) |
| php | function **get_overLoad**( ) |
| cpp | Y_OVERLOAD_enum **get_overLoad**( ) |
| m | -(Y_OVERLOAD_enum) overLoad |
| pas | function **get_overLoad**( ): Integer |
| vb | function **get_overLoad**( ) As Integer |
| cs | int **get_overLoad**( ) |
| java | int **get_overLoad**( ) |
| py | def **get_overLoad**( ) |
| cmd | YVSource **target** **get_overLoad** |

**Returns :**

either `Y_OVERLOAD_FALSE` or `Y_OVERLOAD_TRUE`, according to true if the device is not able to maintaint the requested voltage output

On failure, throws an exception or returns `Y_OVERLOAD_INVALID`.

## vsource→get_regulationFailure() vsource→regulationFailure()[vsource regulationFailure] YVSource

Returns true if the voltage output is too high regarding the requested voltage .

| | |
|---|---|
| js | function **get_regulationFailure**( ) |
| php | function **get_regulationFailure**( ) |
| cpp | Y_REGULATIONFAILURE_enum **get_regulationFailure**( ) |
| m | -(Y_REGULATIONFAILURE_enum) regulationFailure |
| pas | function **get_regulationFailure**( ): Integer |
| vb | function **get_regulationFailure**( ) As Integer |
| cs | int **get_regulationFailure**( ) |
| java | int **get_regulationFailure**( ) |
| py | def **get_regulationFailure**( ) |
| cmd | YVSource **target** **get_regulationFailure** |

**Returns :**

either `Y_REGULATIONFAILURE_FALSE` or `Y_REGULATIONFAILURE_TRUE`, according to true if the voltage output is too high regarding the requested voltage

On failure, throws an exception or returns `Y_REGULATIONFAILURE_INVALID`.

## vsource→get_unit()
## vsource→unit()[vsource unit]

YVSource

Returns the measuring unit for the voltage.

| | |
|---|---|
| js | function **get_unit**( ) |
| php | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | function **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| java | String **get_unit**( ) |
| py | def **get_unit**( ) |
| cmd | YVSource **target** **get_unit** |

**Returns :**
a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## vsource→get_userData() vsource→userData()[vsource userData] YVSource

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## vsource→get_voltage() vsource→voltage()[vsource voltage] YVSource

Returns the voltage output command (mV)

| | |
|---|---|
| js | function **get_voltage**( ) |
| php | function **get_voltage**( ) |
| cpp | int **get_voltage**( ) |
| m | -(int) voltage |
| pas | function **get_voltage**( ): LongInt |
| vb | function **get_voltage**( ) As Integer |
| cs | int **get_voltage**( ) |
| java | int **get_voltage**( ) |
| py | def **get_voltage**( ) |

**Returns :**
an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns `Y_VOLTAGE_INVALID`.

## vsource→isOnline()[vsource isOnline] YVSource

Checks if the function is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

**Returns :**

`true` if the function can be reached, and `false` otherwise

## vsource→isOnline_async() YVSource

Checks if the function is currently reachable, without raising any error (asynchronous version).

js function **isOnline_async**( **callback**, **context**)

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## vsource→load()[vsource load: ] YVSource

Preloads the function cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## vsource→load_async() YVSource

Preloads the function cache with a specified validity duration (asynchronous version).

`js` function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## vsource→nextVSource()[vsource nextVSource] YVSource

Continues the enumeration of voltage sources started using `yFirstVSource()`.

| | |
|---|---|
| js | function **nextVSource**( ) |
| php | function **nextVSource**( ) |
| cpp | YVSource * **nextVSource**( ) |
| m | -(YVSource*) **nextVSource** |
| pas | function **nextVSource**( ): TYVSource |
| vb | function **nextVSource**( ) As YVSource |
| cs | YVSource **nextVSource**( ) |
| java | YVSource **nextVSource**( ) |
| py | def **nextVSource**( ) |

**Returns :**

a pointer to a `YVSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more voltage sources to enumerate.

## vsource→pulse()[vsource pulse: ] YVSource

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

| | |
|---|---|
| js | function **pulse**( **voltage**, **ms_duration**) |
| php | function **pulse**( **$voltage**, **$ms_duration**) |
| cpp | int **pulse**( int **voltage**, int **ms_duration**) |
| m | -(int) **pulse** : (int) **voltage** : (int) **ms_duration** |
| pas | function **pulse**( **voltage**: integer, **ms_duration**: integer): integer |
| vb | function **pulse**( ByVal **voltage** As Integer,<br>ByVal **ms_duration** As Integer) As Integer |
| cs | int **pulse**( int **voltage**, int **ms_duration**) |
| java | int **pulse**( int **voltage**, int **ms_duration**) |
| py | def **pulse**( **voltage**, **ms_duration**) |
| cmd | YVSource **target** **pulse** **voltage** **ms_duration** |

**Parameters :**

| | |
|---|---|
| **voltage** | pulse voltage, in millivolts |
| **ms_duration** | pulse duration, in millisecondes |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## vsource→registerValueCallback()[vsource registerValueCallback: ] YVSource

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | void **registerValueCallback**( YDisplayUpdateCallback **callback**) |
| pas | procedure **registerValueCallback**( **callback**: TGenericUpdateCallback) |
| vb | procedure **registerValueCallback**( ByVal **callback** As GenericUpdateCallback) |
| cs | void **registerValueCallback**( UpdateCallback **callback**) |
| java | void **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |
| m | -(void) **registerValueCallback** : (YFunctionUpdateCallback) **callback** |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## vsource→set_logicalName() vsource→setLogicalName()[vsource setLogicalName: ] YVSource

Changes the logical name of the voltage source.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YVSource **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the voltage source

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## vsource→set_userData() YVSource
## vsource→setUserData()[vsource setUserData: ]

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## vsource→set_voltage() vsource→setVoltage()[vsource setVoltage: ] YVSource

Tunes the device output voltage (milliVolts).

| | |
|---|---|
| js | function **set_voltage**( **newval**) |
| php | function **set_voltage**( **$newval**) |
| cpp | int **set_voltage**( int **newval**) |
| m | -(int) setVoltage : (int) **newval** |
| pas | function **set_voltage**( **newval**: LongInt): integer |
| vb | function **set_voltage**( ByVal **newval** As Integer) As Integer |
| cs | int **set_voltage**( int **newval**) |
| java | int **set_voltage**( int **newval**) |
| py | def **set_voltage**( **newval**) |
| cmd | YVSource **target** **set_voltage** **newval** |

**Parameters :**

**newval** an integer

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## vsource→voltageMove()[vsource voltageMove: ] YVSource

Performs a smooth move at constant speed toward a given value.

| | |
|---|---|
| js | function **voltageMove**( **target**, **ms_duration**) |
| php | function **voltageMove**( **$target**, **$ms_duration**) |
| cpp | int **voltageMove**( int **target**, int **ms_duration**) |
| m | -(int) **voltageMove** : (int) **target** : (int) **ms_duration** |
| pas | function **voltageMove**( **target**: integer, **ms_duration**: integer): integer |
| vb | function **voltageMove**( ByVal **target** As Integer,<br>ByVal **ms_duration** As Integer) As Integer |
| cs | int **voltageMove**( int **target**, int **ms_duration**) |
| java | int **voltageMove**( int **target**, int **ms_duration**) |
| py | def **voltageMove**( **target**, **ms_duration**) |
| cmd | YVSource **target** **voltageMove** **target** **ms_duration** |

**Parameters :**

| | |
|---|---|
| **target** | new output value at end of transition, in milliVolts. |
| **ms_duration** | transition duration, in milliseconds |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## vsource→wait_async() YVSource

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing :

## 3.43. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_wakeupmonitor.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YWakeUpMonitor = yoctolib.YWakeUpMonitor; |
| php | require_once('yocto_wakeupmonitor.php'); |
| cpp | #include "yocto_wakeupmonitor.h" |
| m | #import "yocto_wakeupmonitor.h" |
| pas | uses yocto_wakeupmonitor; |
| vb | yocto_wakeupmonitor.vb |
| cs | yocto_wakeupmonitor.cs |
| java | import com.yoctopuce.YoctoAPI.YWakeUpMonitor; |
| py | from yocto_wakeupmonitor import * |

**Global functions**

**yFindWakeUpMonitor(func)**

Retrieves a monitor for a given identifier.

**yFirstWakeUpMonitor()**

Starts the enumeration of monitors currently accessible.

**`YWakeUpMonitor` methods**

**wakeupmonitor→describe()**

Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**wakeupmonitor→get_advertisedValue()**

Returns the current value of the monitor (no more than 6 characters).

**wakeupmonitor→get_errorMessage()**

Returns the error message of the latest error with the monitor.

**wakeupmonitor→get_errorType()**

Returns the numerical error code of the latest error with the monitor.

**wakeupmonitor→get_friendlyName()**

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

**wakeupmonitor→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**wakeupmonitor→get_functionId()**

Returns the hardware identifier of the monitor, without reference to the module.

**wakeupmonitor→get_hardwareId()**

Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.

**wakeupmonitor→get_logicalName()**

Returns the logical name of the monitor.

**wakeupmonitor→get_module()**

Gets the `YModule` object for the device on which the function is located.

**wakeupmonitor→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**wakeupmonitor→get_nextWakeUp()**

Returns the next scheduled wake up date/time (UNIX format)

**wakeupmonitor→get_powerDuration()**

Returns the maximal wake up time (in seconds) before automatically going to sleep.

**wakeupmonitor→get_sleepCountdown()**

Returns the delay before the next sleep period.

**wakeupmonitor→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**wakeupmonitor→get_wakeUpReason()**

Returns the latest wake up reason.

**wakeupmonitor→get_wakeUpState()**

Returns the current state of the monitor

**wakeupmonitor→isOnline()**

Checks if the monitor is currently reachable, without raising any error.

**wakeupmonitor→isOnline_async(callback, context)**

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

**wakeupmonitor→load(msValidity)**

Preloads the monitor cache with a specified validity duration.

**wakeupmonitor→load_async(msValidity, callback, context)**

Preloads the monitor cache with a specified validity duration (asynchronous version).

**wakeupmonitor→nextWakeUpMonitor()**

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

**wakeupmonitor→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**wakeupmonitor→resetSleepCountDown()**

Resets the sleep countdown.

**wakeupmonitor→set_logicalName(newval)**

Changes the logical name of the monitor.

**wakeupmonitor→set_nextWakeUp(newval)**

Changes the days of the week when a wake up must take place.

**wakeupmonitor→set_powerDuration(newval)**

Changes the maximal wake up time (seconds) before automatically going to sleep.

**wakeupmonitor→set_sleepCountdown(newval)**

Changes the delay before the next sleep period.

**wakeupmonitor→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**wakeupmonitor→sleep(secBeforeSleep)**

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

**wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep)**

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

**wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep)**

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

**wakeupmonitor→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**wakeupmonitor→wakeUp**()

Forces a wake up.

## YWakeUpMonitor.FindWakeUpMonitor() yFindWakeUpMonitor()yFindWakeUpMonitor() YWakeUpMonitor

Retrieves a monitor for a given identifier.

| | |
|---|---|
| js | function **yFindWakeUpMonitor**( **func**) |
| nodejs | function **FindWakeUpMonitor**( **func**) |
| php | function **yFindWakeUpMonitor**( **$func**) |
| cpp | YWakeUpMonitor* **yFindWakeUpMonitor**( const string& **func**) |
| m | YWakeUpMonitor* **yFindWakeUpMonitor**( NSString* **func**) |
| pas | function **yFindWakeUpMonitor**( **func**: string): TYWakeUpMonitor |
| vb | function **yFindWakeUpMonitor**( ByVal **func** As String) As YWakeUpMonitor |
| cs | YWakeUpMonitor **FindWakeUpMonitor**( string **func**) |
| java | YWakeUpMonitor **FindWakeUpMonitor**( String **func**) |
| py | def **FindWakeUpMonitor**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpMonitor.isOnline()` to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the monitor

**Returns :**

a `YWakeUpMonitor` object allowing you to drive the monitor.

## YWakeUpMonitor.FirstWakeUpMonitor() yFirstWakeUpMonitor()yFirstWakeUpMonitor() YWakeUpMonitor

Starts the enumeration of monitors currently accessible.

| | |
|---|---|
| js | function **yFirstWakeUpMonitor**( ) |
| nodejs | function **FirstWakeUpMonitor**( ) |
| php | function **yFirstWakeUpMonitor**( ) |
| cpp | YWakeUpMonitor* **yFirstWakeUpMonitor**( ) |
| m | YWakeUpMonitor* **yFirstWakeUpMonitor**( ) |
| pas | function **yFirstWakeUpMonitor**( ): TYWakeUpMonitor |
| vb | function **yFirstWakeUpMonitor**( ) As YWakeUpMonitor |
| cs | YWakeUpMonitor **FirstWakeUpMonitor**( ) |
| java | YWakeUpMonitor **FirstWakeUpMonitor**( ) |
| py | def **FirstWakeUpMonitor**( ) |

Use the method `YWakeUpMonitor.nextWakeUpMonitor()` to iterate on next monitors.

**Returns :**

a pointer to a `YWakeUpMonitor` object, corresponding to the first monitor currently online, or a `null` pointer if there are none.

## wakeupmonitor→describe()[wakeupmonitor describe] YWakeUpMonitor

Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the monitor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## wakeupmonitor→get_advertisedValue() wakeupmonitor→advertisedValue()[wakeupmonitor advertisedValue] YWakeUpMonitor

Returns the current value of the monitor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YWakeUpMonitor **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the monitor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## wakeupmonitor→get_errorMessage() wakeupmonitor→errorMessage()[wakeupmonitor errorMessage] YWakeUpMonitor

Returns the error message of the latest error with the monitor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the monitor object

## wakeupmonitor→get_errorType() wakeupmonitor→errorType() YWakeUpMonitor

Returns the numerical error code of the latest error with the monitor.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the monitor object

## wakeupmonitor→get_friendlyName() wakeupmonitor→friendlyName()[wakeupmonitor friendlyName]

YWakeUpMonitor

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the monitor if they are defined, otherwise the serial number of the module and the hardware identifier of the monitor (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the monitor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## wakeupmonitor→get_functionDescriptor() wakeupmonitor→functionDescriptor() [wakeupmonitor functionDescriptor] YWakeUpMonitor

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## wakeupmonitor→get_functionId() wakeupmonitor→functionId()[wakeupmonitor functionId] YWakeUpMonitor

Returns the hardware identifier of the monitor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the monitor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## wakeupmonitor→get_hardwareId() wakeupmonitor→hardwareId()[wakeupmonitor hardwareId] YWakeUpMonitor

Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the monitor. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the monitor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

### wakeupmonitor→get_logicalName() wakeupmonitor→logicalName()[wakeupmonitor logicalName] YWakeUpMonitor

Returns the logical name of the monitor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YWakeUpMonitor **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the monitor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## wakeupmonitor→get_module() wakeupmonitor→module()[wakeupmonitor module] YWakeUpMonitor

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## wakeupmonitor→get_module_async() wakeupmonitor→module_async() YWakeUpMonitor

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wakeupmonitor→get_nextWakeUp() wakeupmonitor→nextWakeUp()[wakeupmonitor nextWakeUp] YWakeUpMonitor

Returns the next scheduled wake up date/time (UNIX format)

| | |
|---|---|
| js | function **get_nextWakeUp**( ) |
| nodejs | function **get_nextWakeUp**( ) |
| php | function **get_nextWakeUp**( ) |
| cpp | s64 **get_nextWakeUp**( ) |
| m | -(s64) nextWakeUp |
| pas | function **get_nextWakeUp**( ): int64 |
| vb | function **get_nextWakeUp**( ) As Long |
| cs | long **get_nextWakeUp**( ) |
| java | long **get_nextWakeUp**( ) |
| py | def **get_nextWakeUp**( ) |

**Returns :**
an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns `Y_NEXTWAKEUP_INVALID`.

## wakeupmonitor→get_powerDuration() wakeupmonitor→powerDuration()[wakeupmonitor powerDuration] YWakeUpMonitor

Returns the maximal wake up time (in seconds) before automatically going to sleep.

| | |
|---|---|
| js | function **get_powerDuration**( ) |
| nodejs | function **get_powerDuration**( ) |
| php | function **get_powerDuration**( ) |
| cpp | int **get_powerDuration**( ) |
| m | -(int) powerDuration |
| pas | function **get_powerDuration**( ): LongInt |
| vb | function **get_powerDuration**( ) As Integer |
| cs | int **get_powerDuration**( ) |
| java | int **get_powerDuration**( ) |
| py | def **get_powerDuration**( ) |
| cmd | YWakeUpMonitor **target** **get_powerDuration** |

**Returns :**
an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns `Y_POWERDURATION_INVALID`.

## wakeupmonitor→get_sleepCountdown() wakeupmonitor→sleepCountdown()[wakeupmonitor sleepCountdown] YWakeUpMonitor

Returns the delay before the next sleep period.

| | |
|---|---|
| js | function **get_sleepCountdown**( ) |
| nodejs | function **get_sleepCountdown**( ) |
| php | function **get_sleepCountdown**( ) |
| cpp | int **get_sleepCountdown**( ) |
| m | -(int) sleepCountdown |
| pas | function **get_sleepCountdown**( ): LongInt |
| vb | function **get_sleepCountdown**( ) As Integer |
| cs | int **get_sleepCountdown**( ) |
| java | int **get_sleepCountdown**( ) |
| py | def **get_sleepCountdown**( ) |
| cmd | YWakeUpMonitor **target** **get_sleepCountdown** |

**Returns :**
an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns `Y_SLEEPCOUNTDOWN_INVALID`.

## wakeupmonitor→get_userData() wakeupmonitor→userData()[wakeupmonitor userData] YWakeUpMonitor

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## wakeupmonitor→get_wakeUpReason() wakeupmonitor→wakeUpReason()[wakeupmonitor wakeUpReason] YWakeUpMonitor

Returns the latest wake up reason.

| | |
|---|---|
| js | function **get_wakeUpReason**( ) |
| nodejs | function **get_wakeUpReason**( ) |
| php | function **get_wakeUpReason**( ) |
| cpp | Y_WAKEUPREASON_enum **get_wakeUpReason**( ) |
| m | -(Y_WAKEUPREASON_enum) wakeUpReason |
| pas | function **get_wakeUpReason**( ): Integer |
| vb | function **get_wakeUpReason**( ) As Integer |
| cs | int **get_wakeUpReason**( ) |
| java | int **get_wakeUpReason**( ) |
| py | def **get_wakeUpReason**( ) |
| cmd | YWakeUpMonitor **target** **get_wakeUpReason** |

**Returns :**

a value among Y_WAKEUPREASON_USBPOWER, Y_WAKEUPREASON_EXTPOWER, Y_WAKEUPREASON_ENDOFSLEEP, Y_WAKEUPREASON_EXTSIG1, Y_WAKEUPREASON_EXTSIG2, Y_WAKEUPREASON_EXTSIG3, Y_WAKEUPREASON_EXTSIG4, Y_WAKEUPREASON_SCHEDULE1, Y_WAKEUPREASON_SCHEDULE2, Y_WAKEUPREASON_SCHEDULE3, Y_WAKEUPREASON_SCHEDULE4, Y_WAKEUPREASON_SCHEDULE5 and Y_WAKEUPREASON_SCHEDULE6 corresponding to the latest wake up reason

On failure, throws an exception or returns Y_WAKEUPREASON_INVALID.

## wakeupmonitor→get_wakeUpState() wakeupmonitor→wakeUpState()[wakeupmonitor wakeUpState] YWakeUpMonitor

Returns the current state of the monitor

| | |
|---|---|
| js | function **get_wakeUpState**( ) |
| nodejs | function **get_wakeUpState**( ) |
| php | function **get_wakeUpState**( ) |
| cpp | Y_WAKEUPSTATE_enum **get_wakeUpState**( ) |
| m | -(Y_WAKEUPSTATE_enum) wakeUpState |
| pas | function **get_wakeUpState**( ): Integer |
| vb | function **get_wakeUpState**( ) As Integer |
| cs | int **get_wakeUpState**( ) |
| java | int **get_wakeUpState**( ) |
| py | def **get_wakeUpState**( ) |

**Returns :**

either Y_WAKEUPSTATE_SLEEPING or Y_WAKEUPSTATE_AWAKE, according to the current state of the monitor

On failure, throws an exception or returns Y_WAKEUPSTATE_INVALID.

## wakeupmonitor→isOnline()[wakeupmonitor isOnline] YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

**Returns :**

`true` if the monitor can be reached, and `false` otherwise

## wakeupmonitor→isOnline_async() YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

| | |
|---|---|
| js | function **isOnline_async**( **callback**, **context**) |
| nodejs | function **isOnline_async**( **callback**, **context**) |

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wakeupmonitor→load()[wakeupmonitor load: ] YWakeUpMonitor

Preloads the monitor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupmonitor→load_async() YWakeUpMonitor

Preloads the monitor cache with a specified validity duration (asynchronous version).

js function **load_async**( **msValidity**, **callback**, **context**)
nodejs function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wakeupmonitor→nextWakeUpMonitor() [wakeupmonitor nextWakeUpMonitor] YWakeUpMonitor

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

| | |
|---|---|
| js | function **nextWakeUpMonitor**( ) |
| nodejs | function **nextWakeUpMonitor**( ) |
| php | function **nextWakeUpMonitor**( ) |
| cpp | YWakeUpMonitor * **nextWakeUpMonitor**( ) |
| m | -(YWakeUpMonitor*) **nextWakeUpMonitor** |
| pas | function **nextWakeUpMonitor**( ): TYWakeUpMonitor |
| vb | function **nextWakeUpMonitor**( ) As YWakeUpMonitor |
| cs | YWakeUpMonitor **nextWakeUpMonitor**( ) |
| java | YWakeUpMonitor **nextWakeUpMonitor**( ) |
| py | def **nextWakeUpMonitor**( ) |

**Returns :**

a pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a `null` pointer if there are no more monitors to enumerate.

## wakeupmonitor→registerValueCallback() [wakeupmonitor registerValueCallback: ] YWakeUpMonitor

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YWakeUpMonitorValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YWakeUpMonitorValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYWakeUpMonitorValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## wakeupmonitor→resetSleepCountDown() [wakeupmonitor resetSleepCountDown] YWakeUpMonitor

Resets the sleep countdown.

| | |
|---|---|
| js | function **resetSleepCountDown**( ) |
| nodejs | function **resetSleepCountDown**( ) |
| php | function **resetSleepCountDown**( ) |
| cpp | int **resetSleepCountDown**( ) |
| m | -(int) **resetSleepCountDown** |
| pas | function **resetSleepCountDown**( ): LongInt |
| vb | function **resetSleepCountDown**( ) As Integer |
| cs | int **resetSleepCountDown**( ) |
| java | int **resetSleepCountDown**( ) |
| py | def **resetSleepCountDown**( ) |
| cmd | YWakeUpMonitor **target** **resetSleepCountDown** |

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupmonitor→set_logicalName() wakeupmonitor→setLogicalName()[wakeupmonitor setLogicalName: ] YWakeUpMonitor

Changes the logical name of the monitor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YWakeUpMonitor **target set_logicalName newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the monitor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupmonitor→set_nextWakeUp() wakeupmonitor→setNextWakeUp()[wakeupmonitor setNextWakeUp: ] YWakeUpMonitor

Changes the days of the week when a wake up must take place.

| | |
|---|---|
| js | function **set_nextWakeUp**( **newval**) |
| nodejs | function **set_nextWakeUp**( **newval**) |
| php | function **set_nextWakeUp**( **$newval**) |
| cpp | int **set_nextWakeUp**( s64 **newval**) |
| m | -(int) setNextWakeUp : (s64) **newval** |
| pas | function **set_nextWakeUp**( **newval**: int64): integer |
| vb | function **set_nextWakeUp**( ByVal **newval** As Long) As Integer |
| cs | int **set_nextWakeUp**( long **newval**) |
| java | int **set_nextWakeUp**( long **newval**) |
| py | def **set_nextWakeUp**( **newval**) |
| cmd | YWakeUpMonitor **target** **set_nextWakeUp** **newval** |

**Parameters :**

**newval** an integer corresponding to the days of the week when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupmonitor→set_powerDuration() wakeupmonitor→setPowerDuration()[wakeupmonitor setPowerDuration: ] YWakeUpMonitor

Changes the maximal wake up time (seconds) before automatically going to sleep.

| | |
|---|---|
| js | function **set_powerDuration**( **newval**) |
| nodejs | function **set_powerDuration**( **newval**) |
| php | function **set_powerDuration**( **$newval**) |
| cpp | int **set_powerDuration**( int **newval**) |
| m | -(int) setPowerDuration : (int) **newval** |
| pas | function **set_powerDuration**( **newval**: LongInt): integer |
| vb | function **set_powerDuration**( ByVal **newval** As Integer) As Integer |
| cs | int **set_powerDuration**( int **newval**) |
| java | int **set_powerDuration**( int **newval**) |
| py | def **set_powerDuration**( **newval**) |
| cmd | YWakeUpMonitor **target** **set_powerDuration** **newval** |

**Parameters :**

**newval** an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupmonitor→set_sleepCountdown() wakeupmonitor→setSleepCountdown() [wakeupmonitor setSleepCountdown: ]

YWakeUpMonitor

Changes the delay before the next sleep period.

| | |
|---|---|
| js | function **set_sleepCountdown**( **newval**) |
| nodejs | function **set_sleepCountdown**( **newval**) |
| php | function **set_sleepCountdown**( **$newval**) |
| cpp | int **set_sleepCountdown**( int **newval**) |
| m | -(int) setSleepCountdown : (int) **newval** |
| pas | function **set_sleepCountdown**( **newval**: LongInt): integer |
| vb | function **set_sleepCountdown**( ByVal **newval** As Integer) As Integer |
| cs | int **set_sleepCountdown**( int **newval**) |
| java | int **set_sleepCountdown**( int **newval**) |
| py | def **set_sleepCountdown**( **newval**) |
| cmd | YWakeUpMonitor **target** **set_sleepCountdown** **newval** |

**Parameters :**

**newval** an integer corresponding to the delay before the next sleep period

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupmonitor→set_userData() wakeupmonitor→setUserData()[wakeupmonitor setUserData: ] YWakeUpMonitor

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 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.

| | |
|---|---|
| js | function **sleep**( **secBeforeSleep**) |
| nodejs | function **sleep**( **secBeforeSleep**) |
| php | function **sleep**( **$secBeforeSleep**) |
| cpp | int **sleep**( int **secBeforeSleep**) |
| m | -(int) **sleep** : (int) **secBeforeSleep** |
| pas | function **sleep**( **secBeforeSleep**: LongInt): LongInt |
| vb | function **sleep**( ) As Integer |
| cs | int **sleep**( int **secBeforeSleep**) |
| java | int **sleep**( int **secBeforeSleep**) |
| py | def **sleep**( **secBeforeSleep**) |
| cmd | YWakeUpMonitor **target** **sleep** **secBeforeSleep** |

**Parameters :**

**secBeforeSleep** number of seconds before going into sleep mode,

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupmonitor→sleepFor()[wakeupmonitor sleepFor: ] YWakeUpMonitor

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

| | |
|---|---|
| js | function **sleepFor**( **secUntilWakeUp**, **secBeforeSleep**) |
| nodejs | function **sleepFor**( **secUntilWakeUp**, **secBeforeSleep**) |
| php | function **sleepFor**( **$secUntilWakeUp**, **$secBeforeSleep**) |
| cpp | int **sleepFor**( int **secUntilWakeUp**, int **secBeforeSleep**) |
| m | -(int) **sleepFor** : (int) **secUntilWakeUp** : (int) **secBeforeSleep** |
| pas | function **sleepFor**( **secUntilWakeUp**: LongInt, **secBeforeSleep**: LongInt): LongInt |
| vb | function **sleepFor**( ) As Integer |
| cs | int **sleepFor**( int **secUntilWakeUp**, int **secBeforeSleep**) |
| java | int **sleepFor**( int **secUntilWakeUp**, int **secBeforeSleep**) |
| py | def **sleepFor**( **secUntilWakeUp**, **secBeforeSleep**) |
| cmd | YWakeUpMonitor **target** **sleepFor** **secUntilWakeUp** **secBeforeSleep** |

The count down before sleep can be canceled with resetSleepCountDown.

**Parameters :**

**secUntilWakeUp** sleep duration, in secondes

**secBeforeSleep** number of seconds before going into sleep mode

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupmonitor→sleepUntil()[wakeupmonitor sleepUntil: ] YWakeUpMonitor

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

| | |
|---|---|
| js | function **sleepUntil**( **wakeUpTime**, **secBeforeSleep**) |
| nodejs | function **sleepUntil**( **wakeUpTime**, **secBeforeSleep**) |
| php | function **sleepUntil**( **$wakeUpTime**, **$secBeforeSleep**) |
| cpp | int **sleepUntil**( int **wakeUpTime**, int **secBeforeSleep**) |
| m | -(int) **sleepUntil** : (int) **wakeUpTime** : (int) **secBeforeSleep** |
| pas | function **sleepUntil**( **wakeUpTime**: LongInt, **secBeforeSleep**: LongInt): LongInt |
| vb | function **sleepUntil**( ) As Integer |
| cs | int **sleepUntil**( int **wakeUpTime**, int **secBeforeSleep**) |
| java | int **sleepUntil**( int **wakeUpTime**, int **secBeforeSleep**) |
| py | def **sleepUntil**( **wakeUpTime**, **secBeforeSleep**) |
| cmd | YWakeUpMonitor **target** **sleepUntil** **wakeUpTime** **secBeforeSleep** |

The count down before sleep can be canceled with resetSleepCountDown.

**Parameters :**

**wakeUpTime** wake-up datetime (UNIX format)

**secBeforeSleep** number of seconds before going into sleep mode

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupmonitor→wait_async() YWakeUpMonitor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  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.

| | |
|---|---|
| js | function **wakeUp**( ) |
| nodejs | function **wakeUp**( ) |
| php | function **wakeUp**( ) |
| cpp | int **wakeUp**( ) |
| m | -(int) **wakeUp** |
| pas | function **wakeUp**( ): LongInt |
| vb | function **wakeUp**( ) As Integer |
| cs | int **wakeUp**( ) |
| java | int **wakeUp**( ) |
| py | def **wakeUp**( ) |
| cmd | YWakeUpMonitor **target** **wakeUp** |

# 3.44. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_wakeupschedule.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YWakeUpSchedule = yoctolib.YWakeUpSchedule; |
| php | require_once('yocto_wakeupschedule.php'); |
| cpp | #include "yocto_wakeupschedule.h" |
| m | #import "yocto_wakeupschedule.h" |
| pas | uses yocto_wakeupschedule; |
| vb | yocto_wakeupschedule.vb |
| cs | yocto_wakeupschedule.cs |
| java | import com.yoctopuce.YoctoAPI.YWakeUpSchedule; |
| py | from yocto_wakeupschedule import * |

**Global functions**

**yFindWakeUpSchedule(func)**

Retrieves a wake up schedule for a given identifier.

**yFirstWakeUpSchedule()**

Starts the enumeration of wake up schedules currently accessible.

**`YWakeUpSchedule` methods**

**wakeupschedule→describe()**

Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**wakeupschedule→get_advertisedValue()**

Returns the current value of the wake up schedule (no more than 6 characters).

**wakeupschedule→get_errorMessage()**

Returns the error message of the latest error with the wake up schedule.

**wakeupschedule→get_errorType()**

Returns the numerical error code of the latest error with the wake up schedule.

**wakeupschedule→get_friendlyName()**

Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

**wakeupschedule→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**wakeupschedule→get_functionId()**

Returns the hardware identifier of the wake up schedule, without reference to the module.

**wakeupschedule→get_hardwareId()**

Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.

**wakeupschedule→get_hours()**

Returns the hours scheduled for wake up.

**wakeupschedule→get_logicalName()**

Returns the logical name of the wake up schedule.

**wakeupschedule→get_minutes()**

Returns all the minutes of each hour that are scheduled for wake up.

**wakeupschedule→get_minutesA()**

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

**wakeupschedule→get_minutesB()**

Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

**wakeupschedule→get_module()**

Gets the `YModule` object for the device on which the function is located.

**wakeupschedule→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**wakeupschedule→get_monthDays()**

Returns the days of the month scheduled for wake up.

**wakeupschedule→get_months()**

Returns the months scheduled for wake up.

**wakeupschedule→get_nextOccurence()**

Returns the date/time (seconds) of the next wake up occurence

**wakeupschedule→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**wakeupschedule→get_weekDays()**

Returns the days of the week scheduled for wake up.

**wakeupschedule→isOnline()**

Checks if the wake up schedule is currently reachable, without raising any error.

**wakeupschedule→isOnline_async(callback, context)**

Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

**wakeupschedule→load(msValidity)**

Preloads the wake up schedule cache with a specified validity duration.

**wakeupschedule→load_async(msValidity, callback, context)**

Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

**wakeupschedule→nextWakeUpSchedule()**

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

**wakeupschedule→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**wakeupschedule→set_hours(newval)**

Changes the hours when a wake up must take place.

**wakeupschedule→set_logicalName(newval)**

Changes the logical name of the wake up schedule.

**wakeupschedule→set_minutes(bitmap)**

Changes all the minutes where a wake up must take place.

**wakeupschedule→set_minutesA(newval)**

Changes the minutes in the 00-29 interval when a wake up must take place.

**wakeupschedule→set_minutesB(newval)**

Changes the minutes in the 30-59 interval when a wake up must take place.

**wakeupschedule→set_monthDays(newval)**

Changes the days of the month when a wake up must take place.

**wakeupschedule→set_months(newval)**

Changes the months when a wake up must take place.

**wakeupschedule→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**wakeupschedule→set_weekDays(newval)**

Changes the days of the week when a wake up must take place.

**wakeupschedule→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YWakeUpSchedule.FindWakeUpSchedule() yFindWakeUpSchedule()yFindWakeUpSchedule() YWakeUpSchedule

Retrieves a wake up schedule for a given identifier.

| | |
|---|---|
| js | function **yFindWakeUpSchedule**( **func**) |
| nodejs | function **FindWakeUpSchedule**( **func**) |
| php | function **yFindWakeUpSchedule**( **$func**) |
| cpp | YWakeUpSchedule* **yFindWakeUpSchedule**( const string& **func**) |
| m | YWakeUpSchedule* **yFindWakeUpSchedule**( NSString* **func**) |
| pas | function **yFindWakeUpSchedule**( **func**: string): TYWakeUpSchedule |
| vb | function **yFindWakeUpSchedule**( ByVal **func** As String) As YWakeUpSchedule |
| cs | YWakeUpSchedule **FindWakeUpSchedule**( string **func**) |
| java | YWakeUpSchedule **FindWakeUpSchedule**( String **func**) |
| py | def **FindWakeUpSchedule**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpSchedule.isOnline()` to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the wake up schedule

**Returns :**

a `YWakeUpSchedule` object allowing you to drive the wake up schedule.

## YWakeUpSchedule.FirstWakeUpSchedule() yFirstWakeUpSchedule()yFirstWakeUpSchedule() YWakeUpSchedule

Starts the enumeration of wake up schedules currently accessible.

| | |
|---|---|
| js | function **yFirstWakeUpSchedule**( ) |
| nodejs | function **FirstWakeUpSchedule**( ) |
| php | function **yFirstWakeUpSchedule**( ) |
| cpp | YWakeUpSchedule* **yFirstWakeUpSchedule**( ) |
| m | YWakeUpSchedule* **yFirstWakeUpSchedule**( ) |
| pas | function **yFirstWakeUpSchedule**( ): TYWakeUpSchedule |
| vb | function **yFirstWakeUpSchedule**( ) As YWakeUpSchedule |
| cs | YWakeUpSchedule **FirstWakeUpSchedule**( ) |
| java | YWakeUpSchedule **FirstWakeUpSchedule**( ) |
| py | def **FirstWakeUpSchedule**( ) |

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

**Returns :**

a pointer to a `YWakeUpSchedule` object, corresponding to the first wake up schedule currently online, or a `null` pointer if there are none.

## wakeupschedule→describe()[wakeupschedule describe] YWakeUpSchedule

Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the wake up schedule (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## wakeupschedule→get_advertisedValue() wakeupschedule→advertisedValue() [wakeupschedule advertisedValue] YWakeUpSchedule

Returns the current value of the wake up schedule (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YWakeUpSchedule **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the wake up schedule (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## wakeupschedule→get_errorMessage() wakeupschedule→errorMessage()[wakeupschedule errorMessage] YWakeUpSchedule

Returns the error message of the latest error with the wake up schedule.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the wake up schedule object

## wakeupschedule→get_errorType() wakeupschedule→errorType() YWakeUpSchedule

Returns the numerical error code of the latest error with the wake up schedule.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occured while using the wake up schedule object

## wakeupschedule→get_friendlyName() wakeupschedule→friendlyName()[wakeupschedule friendlyName] YWakeUpSchedule

Returns a global identifier of the wake up schedule in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the wake up schedule if they are defined, otherwise the serial number of the module and the hardware identifier of the wake up schedule (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the wake up schedule using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## wakeupschedule→get_functionDescriptor() wakeupschedule→functionDescriptor() [wakeupschedule functionDescriptor] YWakeUpSchedule

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## wakeupschedule→get_functionId() wakeupschedule→functionId()[wakeupschedule functionId] YWakeUpSchedule

Returns the hardware identifier of the wake up schedule, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the wake up schedule (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## wakeupschedule→get_hardwareId() wakeupschedule→hardwareId()[wakeupschedule hardwareId] YWakeUpSchedule

Returns the unique hardware identifier of the wake up schedule in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wake up schedule. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the wake up schedule (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## wakeupschedule→get_hours() wakeupschedule→hours()[wakeupschedule hours] YWakeUpSchedule

Returns the hours scheduled for wake up.

| | |
|---|---|
| js | function **get_hours**( ) |
| nodejs | function **get_hours**( ) |
| php | function **get_hours**( ) |
| cpp | int **get_hours**( ) |
| m | -(int) hours |
| pas | function **get_hours**( ): LongInt |
| vb | function **get_hours**( ) As Integer |
| cs | int **get_hours**( ) |
| java | int **get_hours**( ) |
| py | def **get_hours**( ) |
| cmd | YWakeUpSchedule **target** **get_hours** |

**Returns :**
an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns `Y_HOURS_INVALID`.

## wakeupschedule→get_logicalName() wakeupschedule→logicalName()[wakeupschedule logicalName] YWakeUpSchedule

Returns the logical name of the wake up schedule.

| | |
|---|---|
| `js` | function **get_logicalName**( ) |
| `nodejs` | function **get_logicalName**( ) |
| `php` | function **get_logicalName**( ) |
| `cpp` | string **get_logicalName**( ) |
| `m` | -(NSString*) logicalName |
| `pas` | function **get_logicalName**( ): string |
| `vb` | function **get_logicalName**( ) As String |
| `cs` | string **get_logicalName**( ) |
| `java` | String **get_logicalName**( ) |
| `py` | def **get_logicalName**( ) |
| `cmd` | YWakeUpSchedule **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the wake up schedule. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## wakeupschedule→get_minutes() wakeupschedule→minutes()[wakeupschedule minutes] YWakeUpSchedule

Returns all the minutes of each hour that are scheduled for wake up.

| | |
|---|---|
| js | function **get_minutes**( ) |
| nodejs | function **get_minutes**( ) |
| php | function **get_minutes**( ) |
| cpp | s64 **get_minutes**( ) |
| m | -(s64) minutes |
| pas | function **get_minutes**( ): int64 |
| vb | function **get_minutes**( ) As Long |
| cs | long **get_minutes**( ) |
| java | long **get_minutes**( ) |
| py | def **get_minutes**( ) |
| cmd | YWakeUpSchedule **target** **get_minutes** |

## wakeupschedule→get_minutesA() wakeupschedule→minutesA()[wakeupschedule minutesA] YWakeUpSchedule

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

| | |
|---|---|
| js | function **get_minutesA**( ) |
| nodejs | function **get_minutesA**( ) |
| php | function **get_minutesA**( ) |
| cpp | int **get_minutesA**( ) |
| m | -(int) minutesA |
| pas | function **get_minutesA**( ): LongInt |
| vb | function **get_minutesA**( ) As Integer |
| cs | int **get_minutesA**( ) |
| java | int **get_minutesA**( ) |
| py | def **get_minutesA**( ) |
| cmd | YWakeUpSchedule **target** **get_minutesA** |

**Returns :**
an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESA_INVALID`.

## wakeupschedule→get_minutesB() wakeupschedule→minutesB()[wakeupschedule minutesB] YWakeUpSchedule

Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

| | |
|---|---|
| js | function **get_minutesB**( ) |
| nodejs | function **get_minutesB**( ) |
| php | function **get_minutesB**( ) |
| cpp | int **get_minutesB**( ) |
| m | -(int) minutesB |
| pas | function **get_minutesB**( ): LongInt |
| vb | function **get_minutesB**( ) As Integer |
| cs | int **get_minutesB**( ) |
| java | int **get_minutesB**( ) |
| py | def **get_minutesB**( ) |
| cmd | YWakeUpSchedule **target** **get_minutesB** |

**Returns :**
an integer corresponding to the minutes in the 30-59 intervalof each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESB_INVALID`.

## wakeupschedule→get_module() wakeupschedule→module()[wakeupschedule module] YWakeUpSchedule

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## wakeupschedule→get_module_async() wakeupschedule→module_async() YWakeUpSchedule

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`js` function **get_module_async**( **callback**, **context**)
`nodejs` function **get_module_async**( **callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wakeupschedule→get_monthDays() wakeupschedule→monthDays()[wakeupschedule monthDays] YWakeUpSchedule

Returns the days of the month scheduled for wake up.

| | |
|---|---|
| js | function **get_monthDays**( ) |
| nodejs | function **get_monthDays**( ) |
| php | function **get_monthDays**( ) |
| cpp | int **get_monthDays**( ) |
| m | -(int) monthDays |
| pas | function **get_monthDays**( ): LongInt |
| vb | function **get_monthDays**( ) As Integer |
| cs | int **get_monthDays**( ) |
| java | int **get_monthDays**( ) |
| py | def **get_monthDays**( ) |
| cmd | YWakeUpSchedule **target** **get_monthDays** |

**Returns :**
an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns `Y_MONTHDAYS_INVALID`.

## wakeupschedule→get_months() wakeupschedule→months()[wakeupschedule months] YWakeUpSchedule

Returns the months scheduled for wake up.

| | |
|---|---|
| js | function **get_months**( ) |
| nodejs | function **get_months**( ) |
| php | function **get_months**( ) |
| cpp | int **get_months**( ) |
| m | -(int) months |
| pas | function **get_months**( ): LongInt |
| vb | function **get_months**( ) As Integer |
| cs | int **get_months**( ) |
| java | int **get_months**( ) |
| py | def **get_months**( ) |
| cmd | YWakeUpSchedule **target** **get_months** |

**Returns :**
an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns `Y_MONTHS_INVALID`.

## wakeupschedule→get_nextOccurence() wakeupschedule→nextOccurence()[wakeupschedule nextOccurence] YWakeUpSchedule

Returns the date/time (seconds) of the next wake up occurence

| | |
|---|---|
| js | function **get_nextOccurence**( ) |
| nodejs | function **get_nextOccurence**( ) |
| php | function **get_nextOccurence**( ) |
| cpp | s64 **get_nextOccurence**( ) |
| m | -(s64) nextOccurence |
| pas | function **get_nextOccurence**( ): int64 |
| vb | function **get_nextOccurence**( ) As Long |
| cs | long **get_nextOccurence**( ) |
| java | long **get_nextOccurence**( ) |
| py | def **get_nextOccurence**( ) |

**Returns :**
an integer corresponding to the date/time (seconds) of the next wake up occurence

On failure, throws an exception or returns `Y_NEXTOCCURENCE_INVALID`.

## wakeupschedule→get_userData() wakeupschedule→userData()[wakeupschedule userData] YWakeUpSchedule

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## wakeupschedule→get_weekDays() wakeupschedule→weekDays()[wakeupschedule weekDays] YWakeUpSchedule

Returns the days of the week scheduled for wake up.

| | |
|---|---|
| js | function **get_weekDays**( ) |
| nodejs | function **get_weekDays**( ) |
| php | function **get_weekDays**( ) |
| cpp | int **get_weekDays**( ) |
| m | -(int) weekDays |
| pas | function **get_weekDays**( ): LongInt |
| vb | function **get_weekDays**( ) As Integer |
| cs | int **get_weekDays**( ) |
| java | int **get_weekDays**( ) |
| py | def **get_weekDays**( ) |
| cmd | YWakeUpSchedule **target** **get_weekDays** |

**Returns :**

an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns `Y_WEEKDAYS_INVALID`.

## wakeupschedule→isOnline()[wakeupschedule isOnline] YWakeUpSchedule

Checks if the wake up schedule is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

**Returns :**

`true` if the wake up schedule can be reached, and `false` otherwise

## wakeupschedule→isOnline_async() YWakeUpSchedule

Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wakeupschedule→load()[wakeupschedule load: ] YWakeUpSchedule

Preloads the wake up schedule cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupschedule→load_async() YWakeUpSchedule

Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

`js` function **load_async**( **msValidity**, **callback**, **context**)
`nodejs` function **load_async**( **msValidity**, **callback**, **context**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wakeupschedule→nextWakeUpSchedule() [wakeupschedule nextWakeUpSchedule] YWakeUpSchedule

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

| | |
|---|---|
| js | function **nextWakeUpSchedule**( ) |
| nodejs | function **nextWakeUpSchedule**( ) |
| php | function **nextWakeUpSchedule**( ) |
| cpp | YWakeUpSchedule * **nextWakeUpSchedule**( ) |
| m | -(YWakeUpSchedule*) **nextWakeUpSchedule** |
| pas | function **nextWakeUpSchedule**( ): TYWakeUpSchedule |
| vb | function **nextWakeUpSchedule**( ) As YWakeUpSchedule |
| cs | YWakeUpSchedule **nextWakeUpSchedule**( ) |
| java | YWakeUpSchedule **nextWakeUpSchedule**( ) |
| py | def **nextWakeUpSchedule**( ) |

**Returns :**

a pointer to a `YWakeUpSchedule` object, corresponding to a wake up schedule currently online, or a `null` pointer if there are no more wake up schedules to enumerate.

## wakeupschedule→registerValueCallback() [wakeupschedule registerValueCallback: ] YWakeUpSchedule

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YWakeUpScheduleValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YWakeUpScheduleValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYWakeUpScheduleValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## wakeupschedule→set_hours() wakeupschedule→setHours()[wakeupschedule setHours: ] YWakeUpSchedule

Changes the hours when a wake up must take place.

| | |
|---|---|
| js | function **set_hours**( **newval**) |
| nodejs | function **set_hours**( **newval**) |
| php | function **set_hours**( **$newval**) |
| cpp | int **set_hours**( int **newval**) |
| m | -(int) setHours : (int) **newval** |
| pas | function **set_hours**( **newval**: LongInt): integer |
| vb | function **set_hours**( ByVal **newval** As Integer) As Integer |
| cs | int **set_hours**( int **newval**) |
| java | int **set_hours**( int **newval**) |
| py | def **set_hours**( **newval**) |
| cmd | YWakeUpSchedule **target** **set_hours** **newval** |

**Parameters :**

**newval** an integer corresponding to the hours when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_logicalName() wakeupschedule→setLogicalName() [wakeupschedule setLogicalName: ]

**YWakeUpSchedule**

Changes the logical name of the wake up schedule.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YWakeUpSchedule **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the wake up schedule.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_minutes() wakeupschedule→setMinutes()[wakeupschedule setMinutes: ] YWakeUpSchedule

Changes all the minutes where a wake up must take place.

| | |
|---|---|
| js | function **set_minutes**( **bitmap**) |
| nodejs | function **set_minutes**( **bitmap**) |
| php | function **set_minutes**( **$bitmap**) |
| cpp | int **set_minutes**( s64 **bitmap**) |
| m | -(int) setMinutes : (s64) **bitmap** |
| pas | function **set_minutes**( **bitmap**: int64): LongInt |
| vb | function **set_minutes**( ) As Integer |
| cs | int **set_minutes**( long **bitmap**) |
| java | int **set_minutes**( long **bitmap**) |
| py | def **set_minutes**( **bitmap**) |
| cmd | YWakeUpSchedule **target** **set_minutes** **bitmap** |

**Parameters :**

**bitmap** Minutes 00-59 of each hour scheduled for wake up.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_minutesA() wakeupschedule→setMinutesA()[wakeupschedule setMinutesA: ] YWakeUpSchedule

Changes the minutes in the 00-29 interval when a wake up must take place.

| | |
|---|---|
| js | function **set_minutesA**( **newval**) |
| nodejs | function **set_minutesA**( **newval**) |
| php | function **set_minutesA**( **$newval**) |
| cpp | int **set_minutesA**( int **newval**) |
| m | -(int) setMinutesA : (int) **newval** |
| pas | function **set_minutesA**( **newval**: LongInt): integer |
| vb | function **set_minutesA**( ByVal **newval** As Integer) As Integer |
| cs | int **set_minutesA**( int **newval**) |
| java | int **set_minutesA**( int **newval**) |
| py | def **set_minutesA**( **newval**) |
| cmd | YWakeUpSchedule **target** **set_minutesA** **newval** |

**Parameters :**

**newval** an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_minutesB() wakeupschedule→setMinutesB()[wakeupschedule setMinutesB: ] YWakeUpSchedule

Changes the minutes in the 30-59 interval when a wake up must take place.

| | |
|---|---|
| js | function **set_minutesB**( **newval**) |
| nodejs | function **set_minutesB**( **newval**) |
| php | function **set_minutesB**( **$newval**) |
| cpp | int **set_minutesB**( int **newval**) |
| m | -(int) setMinutesB : (int) **newval** |
| pas | function **set_minutesB**( **newval**: LongInt): integer |
| vb | function **set_minutesB**( ByVal **newval** As Integer) As Integer |
| cs | int **set_minutesB**( int **newval**) |
| java | int **set_minutesB**( int **newval**) |
| py | def **set_minutesB**( **newval**) |
| cmd | YWakeUpSchedule **target** **set_minutesB** **newval** |

**Parameters :**

**newval** an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_monthDays() wakeupschedule→setMonthDays()[wakeupschedule setMonthDays: ] YWakeUpSchedule

Changes the days of the month when a wake up must take place.

| | |
|---|---|
| js | function **set_monthDays**( **newval**) |
| nodejs | function **set_monthDays**( **newval**) |
| php | function **set_monthDays**( **$newval**) |
| cpp | int **set_monthDays**( int **newval**) |
| m | -(int) setMonthDays : (int) **newval** |
| pas | function **set_monthDays**( **newval**: LongInt): integer |
| vb | function **set_monthDays**( ByVal **newval** As Integer) As Integer |
| cs | int **set_monthDays**( int **newval**) |
| java | int **set_monthDays**( int **newval**) |
| py | def **set_monthDays**( **newval**) |
| cmd | YWakeUpSchedule **target set_monthDays newval** |

**Parameters :**

**newval** an integer corresponding to the days of the month when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_months() wakeupschedule→setMonths()[wakeupschedule setMonths: ] YWakeUpSchedule

Changes the months when a wake up must take place.

| | |
|---|---|
| js | function **set_months**( **newval**) |
| nodejs | function **set_months**( **newval**) |
| php | function **set_months**( **$newval**) |
| cpp | int **set_months**( int **newval**) |
| m | -(int) setMonths : (int) **newval** |
| pas | function **set_months**( **newval**: LongInt): integer |
| vb | function **set_months**( ByVal **newval** As Integer) As Integer |
| cs | int **set_months**( int **newval**) |
| java | int **set_months**( int **newval**) |
| py | def **set_months**( **newval**) |
| cmd | YWakeUpSchedule **target** **set_months** **newval** |

**Parameters :**

**newval** an integer corresponding to the months when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupschedule→set_userData() wakeupschedule→setUserData()[wakeupschedule setUserData: ] YWakeUpSchedule

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## wakeupschedule→set_weekDays() wakeupschedule→setWeekDays()[wakeupschedule setWeekDays: ] YWakeUpSchedule

Changes the days of the week when a wake up must take place.

| | |
|---|---|
| js | function **set_weekDays**( **newval**) |
| nodejs | function **set_weekDays**( **newval**) |
| php | function **set_weekDays**( **$newval**) |
| cpp | int **set_weekDays**( int **newval**) |
| m | -(int) setWeekDays : (int) **newval** |
| pas | function **set_weekDays**( **newval**: LongInt): integer |
| vb | function **set_weekDays**( ByVal **newval** As Integer) As Integer |
| cs | int **set_weekDays**( int **newval**) |
| java | int **set_weekDays**( int **newval**) |
| py | def **set_weekDays**( **newval**) |
| cmd | YWakeUpSchedule **target** **set_weekDays** **newval** |

**Parameters :**

**newval** an integer corresponding to the days of the week when a wake up must take place

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wakeupschedule→wait_async() YWakeUpSchedule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

| | |
|---|---|
| js | function **wait_async**( **callback**, **context**) |
| nodejs | function **wait_async**( **callback**, **context**) |

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.45. Watchdog function interface

The watchog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven direcly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_watchdog.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YWatchdog = yoctolib.YWatchdog; |
| php | require_once('yocto_watchdog.php'); |
| cpp | #include "yocto_watchdog.h" |
| m | #import "yocto_watchdog.h" |
| pas | uses yocto_watchdog; |
| vb | yocto_watchdog.vb |
| cs | yocto_watchdog.cs |
| java | import com.yoctopuce.YoctoAPI.YWatchdog; |
| py | from yocto_watchdog import * |

## Global functions

**yFindWatchdog(func)**

Retrieves a watchdog for a given identifier.

**yFirstWatchdog()**

Starts the enumeration of watchdog currently accessible.

## YWatchdog methods

**watchdog→delayedPulse(ms_delay, ms_duration)**

Schedules a pulse.

**watchdog→describe()**

Returns a short text that describes unambiguously the instance of the watchdog in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**watchdog→get_advertisedValue()**

Returns the current value of the watchdog (no more than 6 characters).

**watchdog→get_autoStart()**

Returns the watchdog runing state at module power up.

**watchdog→get_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

**watchdog→get_errorMessage()**

Returns the error message of the latest error with the watchdog.

**watchdog→get_errorType()**

Returns the numerical error code of the latest error with the watchdog.

**watchdog→get_friendlyName()**

Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.

**watchdog→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**watchdog→get_functionId()**

Returns the hardware identifier of the watchdog, without reference to the module.

**watchdog→get_hardwareId()**

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

**watchdog→get_logicalName()**

Returns the logical name of the watchdog.

**watchdog→get_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

**watchdog→get_maxTimeOnStateB()**

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

**watchdog→get_module()**

Gets the `YModule` object for the device on which the function is located.

**watchdog→get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

**watchdog→get_output()**

Returns the output state of the watchdog, when used as a simple switch (single throw).

**watchdog→get_pulseTimer()**

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

**watchdog→get_running()**

Returns the watchdog running state.

**watchdog→get_state()**

Returns the state of the watchdog (A for the idle position, B for the active position).

**watchdog→get_stateAtPowerOn()**

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

**watchdog→get_triggerDelay()**

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

**watchdog→get_triggerDuration()**

Returns the duration of resets caused by the watchdog, in milliseconds.

**watchdog→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**watchdog→isOnline()**

Checks if the watchdog is currently reachable, without raising any error.

**watchdog→isOnline_async(callback, context)**

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

**watchdog→load(msValidity)**

Preloads the watchdog cache with a specified validity duration.

**watchdog→load_async(msValidity, callback, context)**

Preloads the watchdog cache with a specified validity duration (asynchronous version).

**watchdog→nextWatchdog()**

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

**watchdog→pulse(ms_duration)**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

**watchdog→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**watchdog→resetWatchdog()**

Resets the watchdog.

**watchdog→set_autoStart(newval)**

Changes the watchdog runningsttae at module power up.

**watchdog→set_logicalName(newval)**

Changes the logical name of the watchdog.

**watchdog→set_maxTimeOnStateA(newval)**

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

**watchdog→set_maxTimeOnStateB(newval)**

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

**watchdog→set_output(newval)**

Changes the output state of the watchdog, when used as a simple switch (single throw).

**watchdog→set_running(newval)**

Changes the running state of the watchdog.

**watchdog→set_state(newval)**

Changes the state of the watchdog (A for the idle position, B for the active position).

**watchdog→set_stateAtPowerOn(newval)**

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

**watchdog→set_triggerDelay(newval)**

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

**watchdog→set_triggerDuration(newval)**

Changes the duration of resets caused by the watchdog, in milliseconds.

**watchdog→set_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**watchdog→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YWatchdog.FindWatchdog() yFindWatchdog()yFindWatchdog() YWatchdog

Retrieves a watchdog for a given identifier.

| | |
|---|---|
| js | function **yFindWatchdog**( **func**) |
| nodejs | function **FindWatchdog**( **func**) |
| php | function **yFindWatchdog**( **$func**) |
| cpp | YWatchdog* **yFindWatchdog**( const string& **func**) |
| m | YWatchdog* **yFindWatchdog**( NSString* **func**) |
| pas | function **yFindWatchdog**( **func**: string): TYWatchdog |
| vb | function **yFindWatchdog**( ByVal **func** As String) As YWatchdog |
| cs | YWatchdog **FindWatchdog**( string **func**) |
| java | YWatchdog **FindWatchdog**( String **func**) |
| py | def **FindWatchdog**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWatchdog.isOnline()` to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the watchdog

**Returns :**

a `YWatchdog` object allowing you to drive the watchdog.

## YWatchdog.FirstWatchdog() yFirstWatchdog()yFirstWatchdog() YWatchdog

Starts the enumeration of watchdog currently accessible.

| | |
|---|---|
| js | function **yFirstWatchdog**( ) |
| nodejs | function **FirstWatchdog**( ) |
| php | function **yFirstWatchdog**( ) |
| cpp | YWatchdog* **yFirstWatchdog**( ) |
| m | YWatchdog* **yFirstWatchdog**( ) |
| pas | function **yFirstWatchdog**( ): TYWatchdog |
| vb | function **yFirstWatchdog**( ) As YWatchdog |
| cs | YWatchdog **FirstWatchdog**( ) |
| java | YWatchdog **FirstWatchdog**( ) |
| py | def **FirstWatchdog**( ) |

Use the method `YWatchdog.nextWatchdog()` to iterate on next watchdog.

**Returns :**

a pointer to a `YWatchdog` object, corresponding to the first watchdog currently online, or a `null` pointer if there are none.

## watchdog→delayedPulse()[watchdog delayedPulse: ] YWatchdog

Schedules a pulse.

| | |
|---|---|
| js | function **delayedPulse**( **ms_delay**, **ms_duration**) |
| nodejs | function **delayedPulse**( **ms_delay**, **ms_duration**) |
| php | function **delayedPulse**( **$ms_delay**, **$ms_duration**) |
| cpp | int **delayedPulse**( int **ms_delay**, int **ms_duration**) |
| m | -(int) **delayedPulse** : (int) **ms_delay** : (int) **ms_duration** |
| pas | function **delayedPulse**( **ms_delay**: LongInt, **ms_duration**: LongInt): integer |
| vb | function **delayedPulse**( ByVal **ms_delay** As Integer,<br>ByVal **ms_duration** As Integer) As Integer |
| cs | int **delayedPulse**( int **ms_delay**, int **ms_duration**) |
| java | int **delayedPulse**( int **ms_delay**, int **ms_duration**) |
| py | def **delayedPulse**( **ms_delay**, **ms_duration**) |
| cmd | YWatchdog **target** **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`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the watchdog (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## watchdog→get_advertisedValue() watchdog→advertisedValue()[watchdog advertisedValue] YWatchdog

Returns the current value of the watchdog (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YWatchdog **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the watchdog (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## watchdog→get_autoStart() watchdog→autoStart()[watchdog autoStart] YWatchdog

Returns the watchdog runing state at module power up.

| | |
|---|---|
| js | function **get_autoStart**( ) |
| nodejs | function **get_autoStart**( ) |
| php | function **get_autoStart**( ) |
| cpp | Y_AUTOSTART_enum **get_autoStart**( ) |
| m | -(Y_AUTOSTART_enum) autoStart |
| pas | function **get_autoStart**( ): Integer |
| vb | function **get_autoStart**( ) As Integer |
| cs | int **get_autoStart**( ) |
| java | int **get_autoStart**( ) |
| py | def **get_autoStart**( ) |
| cmd | YWatchdog **target** **get_autoStart** |

**Returns :**

either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the watchdog runing state at module power up

On failure, throws an exception or returns `Y_AUTOSTART_INVALID`.

## watchdog→get_countdown() watchdog→countdown()[watchdog countdown] YWatchdog

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

| | |
|---|---|
| js | function **get_countdown**( ) |
| nodejs | function **get_countdown**( ) |
| php | function **get_countdown**( ) |
| cpp | s64 **get_countdown**( ) |
| m | -(s64) countdown |
| pas | function **get_countdown**( ): int64 |
| vb | function **get_countdown**( ) As Long |
| cs | long **get_countdown**( ) |
| java | long **get_countdown**( ) |
| py | def **get_countdown**( ) |
| cmd | YWatchdog **target** **get_countdown** |

**Returns :**

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

## watchdog→get_errorMessage() watchdog→errorMessage()[watchdog errorMessage] YWatchdog

Returns the error message of the latest error with the watchdog.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the watchdog object

## watchdog→get_errorType()
## watchdog→errorType()

**YWatchdog**

Returns the numerical error code of the latest error with the watchdog.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the watchdog object

## watchdog→get_friendlyName() watchdog→friendlyName()[watchdog friendlyName] YWatchdog

Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the watchdog if they are defined, otherwise the serial number of the module and the hardware identifier of the watchdog (for exemple: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the watchdog using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## watchdog→get_functionDescriptor() watchdog→functionDescriptor()[watchdog functionDescriptor] YWatchdog

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**
an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## watchdog→get_functionId()
## watchdog→functionId()[watchdog functionId]

**YWatchdog**

Returns the hardware identifier of the watchdog, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the watchdog (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## watchdog→get_hardwareId() watchdog→hardwareId()[watchdog hardwareId] YWatchdog

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the watchdog (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## watchdog→get_logicalName() watchdog→logicalName()[watchdog logicalName] YWatchdog

Returns the logical name of the watchdog.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YWatchdog **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the watchdog. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## watchdog→get_maxTimeOnStateA() watchdog→maxTimeOnStateA()[watchdog maxTimeOnStateA] YWatchdog

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

| | |
|---|---|
| js | function **get_maxTimeOnStateA**( ) |
| nodejs | function **get_maxTimeOnStateA**( ) |
| php | function **get_maxTimeOnStateA**( ) |
| cpp | s64 **get_maxTimeOnStateA**( ) |
| m | -(s64) maxTimeOnStateA |
| pas | function **get_maxTimeOnStateA**( ): int64 |
| vb | function **get_maxTimeOnStateA**( ) As Long |
| cs | long **get_maxTimeOnStateA**( ) |
| java | long **get_maxTimeOnStateA**( ) |
| py | def **get_maxTimeOnStateA**( ) |
| cmd | YWatchdog **target** **get_maxTimeOnStateA** |

Zero means no maximum time.

**Returns :**
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

## watchdog→get_maxTimeOnStateB() watchdog→maxTimeOnStateB()[watchdog maxTimeOnStateB] YWatchdog

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

| | |
|---|---|
| js | function **get_maxTimeOnStateB**( ) |
| nodejs | function **get_maxTimeOnStateB**( ) |
| php | function **get_maxTimeOnStateB**( ) |
| cpp | s64 **get_maxTimeOnStateB**( ) |
| m | -(s64) maxTimeOnStateB |
| pas | function **get_maxTimeOnStateB**( ): int64 |
| vb | function **get_maxTimeOnStateB**( ) As Long |
| cs | long **get_maxTimeOnStateB**( ) |
| java | long **get_maxTimeOnStateB**( ) |
| py | def **get_maxTimeOnStateB**( ) |
| cmd | YWatchdog **target** **get_maxTimeOnStateB** |

Zero means no maximum time.

**Returns :**
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

## watchdog→get_module() watchdog→module()[watchdog module] YWatchdog

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

## watchdog→get_module_async()
## watchdog→module_async()

**YWatchdog**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

| | |
|---|---|
| js | function **get_module_async**( **callback**, **context**) |
| nodejs | function **get_module_async**( **callback**, **context**) |

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## watchdog→get_output() watchdog→output()[watchdog output] YWatchdog

Returns the output state of the watchdog, when used as a simple switch (single throw).

| | |
|---|---|
| js | function **get_output**( ) |
| nodejs | function **get_output**( ) |
| php | function **get_output**( ) |
| cpp | Y_OUTPUT_enum **get_output**( ) |
| m | -(Y_OUTPUT_enum) output |
| pas | function **get_output**( ): Integer |
| vb | function **get_output**( ) As Integer |
| cs | int **get_output**( ) |
| java | int **get_output**( ) |
| py | def **get_output**( ) |
| cmd | YWatchdog **target** **get_output** |

**Returns :**

either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns `Y_OUTPUT_INVALID`.

## watchdog→get_pulseTimer() watchdog→pulseTimer()[watchdog pulseTimer] YWatchdog

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

| | |
|---|---|
| js | function **get_pulseTimer**( ) |
| nodejs | function **get_pulseTimer**( ) |
| php | function **get_pulseTimer**( ) |
| cpp | s64 **get_pulseTimer**( ) |
| m | -(s64) pulseTimer |
| pas | function **get_pulseTimer**( ): int64 |
| vb | function **get_pulseTimer**( ) As Long |
| cs | long **get_pulseTimer**( ) |
| java | long **get_pulseTimer**( ) |
| py | def **get_pulseTimer**( ) |
| cmd | YWatchdog **target** **get_pulseTimer** |

When there is no ongoing pulse, returns zero.

**Returns :**

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

## watchdog→get_running() watchdog→running()[watchdog running] YWatchdog

Returns the watchdog running state.

| | |
|---|---|
| js | function **get_running**( ) |
| nodejs | function **get_running**( ) |
| php | function **get_running**( ) |
| cpp | Y_RUNNING_enum **get_running**( ) |
| m | -(Y_RUNNING_enum) running |
| pas | function **get_running**( ): Integer |
| vb | function **get_running**( ) As Integer |
| cs | int **get_running**( ) |
| java | int **get_running**( ) |
| py | def **get_running**( ) |
| cmd | YWatchdog **target** **get_running** |

**Returns :**
either `Y_RUNNING_OFF` or `Y_RUNNING_ON`, according to the watchdog running state

On failure, throws an exception or returns `Y_RUNNING_INVALID`.

## watchdog→get_state() watchdog→state()[watchdog state] YWatchdog

Returns the state of the watchdog (A for the idle position, B for the active position).

| | |
|---|---|
| js | function **get_state**( ) |
| nodejs | function **get_state**( ) |
| php | function **get_state**( ) |
| cpp | Y_STATE_enum **get_state**( ) |
| m | -(Y_STATE_enum) state |
| pas | function **get_state**( ): Integer |
| vb | function **get_state**( ) As Integer |
| cs | int **get_state**( ) |
| java | int **get_state**( ) |
| py | def **get_state**( ) |
| cmd | YWatchdog **target** **get_state** |

**Returns :**

either `Y_STATE_A` or `Y_STATE_B`, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns `Y_STATE_INVALID`.

## watchdog→get_stateAtPowerOn() watchdog→stateAtPowerOn()[watchdog stateAtPowerOn] YWatchdog

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

| | |
|---|---|
| js | function **get_stateAtPowerOn**( ) |
| nodejs | function **get_stateAtPowerOn**( ) |
| php | function **get_stateAtPowerOn**( ) |
| cpp | Y_STATEATPOWERON_enum **get_stateAtPowerOn**( ) |
| m | -(Y_STATEATPOWERON_enum) stateAtPowerOn |
| pas | function **get_stateAtPowerOn**( ): Integer |
| vb | function **get_stateAtPowerOn**( ) As Integer |
| cs | int **get_stateAtPowerOn**( ) |
| java | int **get_stateAtPowerOn**( ) |
| py | def **get_stateAtPowerOn**( ) |
| cmd | YWatchdog **target** **get_stateAtPowerOn** |

**Returns :**

a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B` corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns `Y_STATEATPOWERON_INVALID`.

## watchdog→get_triggerDelay() watchdog→triggerDelay()[watchdog triggerDelay] YWatchdog

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

| | |
|---|---|
| js | function **get_triggerDelay**( ) |
| nodejs | function **get_triggerDelay**( ) |
| php | function **get_triggerDelay**( ) |
| cpp | s64 **get_triggerDelay**( ) |
| m | -(s64) triggerDelay |
| pas | function **get_triggerDelay**( ): int64 |
| vb | function **get_triggerDelay**( ) As Long |
| cs | long **get_triggerDelay**( ) |
| java | long **get_triggerDelay**( ) |
| py | def **get_triggerDelay**( ) |
| cmd | YWatchdog **target** **get_triggerDelay** |

**Returns :**

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDELAY_INVALID`.

## watchdog→get_triggerDuration() watchdog→triggerDuration()[watchdog triggerDuration] YWatchdog

Returns the duration of resets caused by the watchdog, in milliseconds.

| | |
|---|---|
| js | function **get_triggerDuration**( ) |
| nodejs | function **get_triggerDuration**( ) |
| php | function **get_triggerDuration**( ) |
| cpp | s64 **get_triggerDuration**( ) |
| m | -(s64) triggerDuration |
| pas | function **get_triggerDuration**( ): int64 |
| vb | function **get_triggerDuration**( ) As Long |
| cs | long **get_triggerDuration**( ) |
| java | long **get_triggerDuration**( ) |
| py | def **get_triggerDuration**( ) |
| cmd | YWatchdog **target** **get_triggerDuration** |

**Returns :**

an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDURATION_INVALID`.

## watchdog→get_userData() watchdog→userData()[watchdog userData] YWatchdog

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| nodejs | function **get_userData**( ) |
| php | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(void*) userData |
| pas | function **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## watchdog→isOnline()[watchdog isOnline] YWatchdog

Checks if the watchdog is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

**Returns :**

`true` if the watchdog can be reached, and `false` otherwise

## watchdog→isOnline_async() YWatchdog

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## watchdog→load()[watchdog load: ] YWatchdog

Preloads the watchdog cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## watchdog→load_async() YWatchdog

Preloads the watchdog cache with a specified validity duration (asynchronous version).

| | |
|---|---|
| js | function **load_async**( **msValidity**, **callback**, **context**) |
| nodejs | function **load_async**( **msValidity**, **callback**, **context**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## watchdog→nextWatchdog()[watchdog nextWatchdog] YWatchdog

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

| | |
|---|---|
| js | function **nextWatchdog**( ) |
| nodejs | function **nextWatchdog**( ) |
| php | function **nextWatchdog**( ) |
| cpp | YWatchdog * **nextWatchdog**( ) |
| m | -(YWatchdog*) **nextWatchdog** |
| pas | function **nextWatchdog**( ): TYWatchdog |
| vb | function **nextWatchdog**( ) As YWatchdog |
| cs | YWatchdog **nextWatchdog**( ) |
| java | YWatchdog **nextWatchdog**( ) |
| py | def **nextWatchdog**( ) |

**Returns :**

a pointer to a `YWatchdog` object, corresponding to a watchdog currently online, or a `null` pointer if there are no more watchdog to enumerate.

## watchdog→pulse()[watchdog pulse: ] YWatchdog

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

| | |
|---|---|
| js | function **pulse**( **ms_duration**) |
| nodejs | function **pulse**( **ms_duration**) |
| php | function **pulse**( **$ms_duration**) |
| cpp | int **pulse**( int **ms_duration**) |
| m | -(int) **pulse** : (int) **ms_duration** |
| pas | function **pulse**( **ms_duration**: LongInt): integer |
| vb | function **pulse**( ByVal **ms_duration** As Integer) As Integer |
| cs | int **pulse**( int **ms_duration**) |
| java | int **pulse**( int **ms_duration**) |
| py | def **pulse**( **ms_duration**) |
| cmd | YWatchdog **target** **pulse** **ms_duration** |

**Parameters :**

**ms_duration** pulse duration, in millisecondes

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→registerValueCallback()[watchdog registerValueCallback: ] YWatchdog

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YWatchdogValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YWatchdogValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYWatchdogValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## watchdog→resetWatchdog()[watchdog resetWatchdog] YWatchdog

Resets the watchdog.

| | |
|---|---|
| js | function **resetWatchdog**( ) |
| nodejs | function **resetWatchdog**( ) |
| php | function **resetWatchdog**( ) |
| cpp | int **resetWatchdog**( ) |
| m | -(int) **resetWatchdog** |
| pas | function **resetWatchdog**( ): integer |
| vb | function **resetWatchdog**( ) As Integer |
| cs | int **resetWatchdog**( ) |
| java | int **resetWatchdog**( ) |
| py | def **resetWatchdog**( ) |
| cmd | YWatchdog **target** **resetWatchdog** |

When the watchdog is running, this function must be called on a regular basis to prevent the watchog to trigger

**Returns :**
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_autoStart() watchdog→setAutoStart()[watchdog setAutoStart: ] YWatchdog

Changes the watchdog runningsttae at module power up.

| | |
|---|---|
| js | function **set_autoStart**( **newval**) |
| nodejs | function **set_autoStart**( **newval**) |
| php | function **set_autoStart**( **$newval**) |
| cpp | int **set_autoStart**( Y_AUTOSTART_enum **newval**) |
| m | -(int) setAutoStart : (Y_AUTOSTART_enum) **newval** |
| pas | function **set_autoStart**( **newval**: Integer): integer |
| vb | function **set_autoStart**( ByVal **newval** As Integer) As Integer |
| cs | int **set_autoStart**( int **newval**) |
| java | int **set_autoStart**( int **newval**) |
| py | def **set_autoStart**( **newval**) |
| cmd | YWatchdog **target** **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 up

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_logicalName() watchdog→setLogicalName()[watchdog setLogicalName: ] YWatchdog

Changes the logical name of the watchdog.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YWatchdog **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the watchdog.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

### watchdog→set_maxTimeOnStateA() watchdog→setMaxTimeOnStateA()[watchdog setMaxTimeOnStateA: ] YWatchdog

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

| | |
|---|---|
| js | function **set_maxTimeOnStateA**( **newval**) |
| nodejs | function **set_maxTimeOnStateA**( **newval**) |
| php | function **set_maxTimeOnStateA**( **$newval**) |
| cpp | int **set_maxTimeOnStateA**( s64 **newval**) |
| m | -(int) setMaxTimeOnStateA : (s64) **newval** |
| pas | function **set_maxTimeOnStateA**( **newval**: int64): integer |
| vb | function **set_maxTimeOnStateA**( ByVal **newval** As Long) As Integer |
| cs | int **set_maxTimeOnStateA**( long **newval**) |
| java | int **set_maxTimeOnStateA**( long **newval**) |
| py | def **set_maxTimeOnStateA**( **newval**) |
| cmd | YWatchdog **target set_maxTimeOnStateA newval** |

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

### watchdog→set_maxTimeOnStateB() watchdog→setMaxTimeOnStateB()[watchdog setMaxTimeOnStateB: ] YWatchdog

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

| | |
|---|---|
| js | function **set_maxTimeOnStateB**( **newval**) |
| nodejs | function **set_maxTimeOnStateB**( **newval**) |
| php | function **set_maxTimeOnStateB**( **$newval**) |
| cpp | int **set_maxTimeOnStateB**( s64 **newval**) |
| m | -(int) setMaxTimeOnStateB : (s64) **newval** |
| pas | function **set_maxTimeOnStateB**( **newval**: int64): integer |
| vb | function **set_maxTimeOnStateB**( ByVal **newval** As Long) As Integer |
| cs | int **set_maxTimeOnStateB**( long **newval**) |
| java | int **set_maxTimeOnStateB**( long **newval**) |
| py | def **set_maxTimeOnStateB**( **newval**) |
| cmd | YWatchdog **target** **set_maxTimeOnStateB** **newval** |

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_output() watchdog→setOutput()[watchdog setOutput: ] YWatchdog

Changes the output state of the watchdog, when used as a simple switch (single throw).

| | |
|---|---|
| js | function **set_output**( **newval**) |
| nodejs | function **set_output**( **newval**) |
| php | function **set_output**( **$newval**) |
| cpp | int **set_output**( Y_OUTPUT_enum **newval**) |
| m | -(int) setOutput : (Y_OUTPUT_enum) **newval** |
| pas | function **set_output**( **newval**: Integer): integer |
| vb | function **set_output**( ByVal **newval** As Integer) As Integer |
| cs | int **set_output**( int **newval**) |
| java | int **set_output**( int **newval**) |
| py | def **set_output**( **newval**) |
| cmd | YWatchdog **target** **set_output** **newval** |

**Parameters :**

**newval** either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the watchdog, when used as a simple switch (single throw)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_running() watchdog→setRunning()[watchdog setRunning: ]

**YWatchdog**

Changes the running state of the watchdog.

| | |
|---|---|
| js | function **set_running**( **newval**) |
| nodejs | function **set_running**( **newval**) |
| php | function **set_running**( **$newval**) |
| cpp | int **set_running**( Y_RUNNING_enum **newval**) |
| m | -(int) setRunning : (Y_RUNNING_enum) **newval** |
| pas | function **set_running**( **newval**: Integer): integer |
| vb | function **set_running**( ByVal **newval** As Integer) As Integer |
| cs | int **set_running**( int **newval**) |
| java | int **set_running**( int **newval**) |
| py | def **set_running**( **newval**) |
| cmd | YWatchdog **target** **set_running** **newval** |

**Parameters :**

**newval** either `Y_RUNNING_OFF` or `Y_RUNNING_ON`, according to the running state of the watchdog

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_state() watchdog→setState()[watchdog setState: ] YWatchdog

Changes the state of the watchdog (A for the idle position, B for the active position).

| | |
|---|---|
| js | function **set_state**( **newval**) |
| nodejs | function **set_state**( **newval**) |
| php | function **set_state**( **$newval**) |
| cpp | int **set_state**( Y_STATE_enum **newval**) |
| m | -(int) setState : (Y_STATE_enum) **newval** |
| pas | function **set_state**( **newval**: Integer): integer |
| vb | function **set_state**( ByVal **newval** As Integer) As Integer |
| cs | int **set_state**( int **newval**) |
| java | int **set_state**( int **newval**) |
| py | def **set_state**( **newval**) |
| cmd | YWatchdog **target** **set_state** **newval** |

**Parameters :**

**newval** either `Y_STATE_A` or `Y_STATE_B`, according to the state of the watchdog (A for the idle position, B for the active position)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

### watchdog→set_stateAtPowerOn() watchdog→setStateAtPowerOn()[watchdog setStateAtPowerOn: ] YWatchdog

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

| | |
|---|---|
| js | function **set_stateAtPowerOn**( **newval**) |
| nodejs | function **set_stateAtPowerOn**( **newval**) |
| php | function **set_stateAtPowerOn**( **$newval**) |
| cpp | int **set_stateAtPowerOn**( Y_STATEATPOWERON_enum **newval**) |
| m | -(int) setStateAtPowerOn : (Y_STATEATPOWERON_enum) **newval** |
| pas | function **set_stateAtPowerOn**( **newval**: Integer): integer |
| vb | function **set_stateAtPowerOn**( ByVal **newval** As Integer) As Integer |
| cs | int **set_stateAtPowerOn**( int **newval**) |
| java | int **set_stateAtPowerOn**( int **newval**) |
| py | def **set_stateAtPowerOn**( **newval**) |
| cmd | YWatchdog **target** **set_stateAtPowerOn** **newval** |

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_triggerDelay() watchdog→setTriggerDelay()[watchdog setTriggerDelay: ] YWatchdog

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

| | |
|---|---|
| js | function **set_triggerDelay**( **newval**) |
| nodejs | function **set_triggerDelay**( **newval**) |
| php | function **set_triggerDelay**( **$newval**) |
| cpp | int **set_triggerDelay**( s64 **newval**) |
| m | -(int) setTriggerDelay : (s64) **newval** |
| pas | function **set_triggerDelay**( **newval**: int64): integer |
| vb | function **set_triggerDelay**( ByVal **newval** As Long) As Integer |
| cs | int **set_triggerDelay**( long **newval**) |
| java | int **set_triggerDelay**( long **newval**) |
| py | def **set_triggerDelay**( **newval**) |
| cmd | YWatchdog **target** **set_triggerDelay** **newval** |

**Parameters :**

**newval** an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_triggerDuration() watchdog→setTriggerDuration()[watchdog setTriggerDuration: ] YWatchdog

Changes the duration of resets caused by the watchdog, in milliseconds.

| | |
|---|---|
| js | function **set_triggerDuration**( **newval**) |
| nodejs | function **set_triggerDuration**( **newval**) |
| php | function **set_triggerDuration**( **$newval**) |
| cpp | int **set_triggerDuration**( s64 **newval**) |
| m | -(int) setTriggerDuration : (s64) **newval** |
| pas | function **set_triggerDuration**( **newval**: int64): integer |
| vb | function **set_triggerDuration**( ByVal **newval** As Long) As Integer |
| cs | int **set_triggerDuration**( long **newval**) |
| java | int **set_triggerDuration**( long **newval**) |
| py | def **set_triggerDuration**( **newval**) |
| cmd | YWatchdog **target set_triggerDuration newval** |

**Parameters :**

**newval** an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set_userData() watchdog→setUserData()[watchdog setUserData: ] YWatchdog

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## watchdog→wait_async() YWatchdog

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

`js` function **wait_async**( **callback**, **context**)
`nodejs` function **wait_async**( **callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 3.46. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_wireless.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YWireless = yoctolib.YWireless; |
| php | require_once('yocto_wireless.php'); |
| cpp | #include "yocto_wireless.h" |
| m | #import "yocto_wireless.h" |
| pas | uses yocto_wireless; |
| vb | yocto_wireless.vb |
| cs | yocto_wireless.cs |
| java | import com.yoctopuce.YoctoAPI.YWireless; |
| py | from yocto_wireless import * |

**Global functions**

**yFindWireless(func)**

Retrieves a wireless lan interface for a given identifier.

**yFirstWireless()**

Starts the enumeration of wireless lan interfaces currently accessible.

**`YWireless` methods**

**wireless→adhocNetwork(ssid, securityKey)**

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

**wireless→describe()**

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**wireless→get_advertisedValue()**

Returns the current value of the wireless lan interface (no more than 6 characters).

**wireless→get_channel()**

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

**wireless→get_detectedWlans()**

Returns a list of YWlanRecord objects that describe detected Wireless networks.

**wireless→get_errorMessage()**

Returns the error message of the latest error with the wireless lan interface.

**wireless→get_errorType()**

Returns the numerical error code of the latest error with the wireless lan interface.

**wireless→get_friendlyName()**

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

**wireless→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**wireless→get_functionId()**

Returns the hardware identifier of the wireless lan interface, without reference to the module.

**wireless→get_hardwareId()**

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

**wireless→get_linkQuality()**
Returns the link quality, expressed in percent.

**wireless→get_logicalName()**
Returns the logical name of the wireless lan interface.

**wireless→get_message()**
Returns the latest status message from the wireless interface.

**wireless→get_module()**
Gets the `YModule` object for the device on which the function is located.

**wireless→get_module_async(callback, context)**
Gets the `YModule` object for the device on which the function is located (asynchronous version).

**wireless→get_security()**
Returns the security algorithm used by the selected wireless network.

**wireless→get_ssid()**
Returns the wireless network name (SSID).

**wireless→get_userData()**
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**wireless→isOnline()**
Checks if the wireless lan interface is currently reachable, without raising any error.

**wireless→isOnline_async(callback, context)**
Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

**wireless→joinNetwork(ssid, securityKey)**
Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

**wireless→load(msValidity)**
Preloads the wireless lan interface cache with a specified validity duration.

**wireless→load_async(msValidity, callback, context)**
Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

**wireless→nextWireless()**
Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

**wireless→registerValueCallback(callback)**
Registers the callback function that is invoked on every change of advertised value.

**wireless→set_logicalName(newval)**
Changes the logical name of the wireless lan interface.

**wireless→set_userData(data)**
Stores a user context provided as argument in the userData attribute of the function.

**wireless→wait_async(callback, context)**
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YWireless.FindWireless() yFindWireless()yFindWireless() YWireless

Retrieves a wireless lan interface for a given identifier.

| | |
|---|---|
| js | function **yFindWireless**( **func**) |
| nodejs | function **FindWireless**( **func**) |
| php | function **yFindWireless**( **$func**) |
| cpp | YWireless* **yFindWireless**( string **func**) |
| m | +(YWireless*) **yFindWireless** : (NSString*) **func** |
| pas | function **yFindWireless**( **func**: string): TYWireless |
| vb | function **yFindWireless**( ByVal **func** As String) As YWireless |
| cs | YWireless **FindWireless**( string **func**) |
| java | YWireless **FindWireless**( String **func**) |
| py | def **FindWireless**( **func**) |

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wireless lan interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWireless.isOnline()` to test if the wireless lan interface is indeed online at a given time. In case of ambiguity when looking for a wireless lan interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the wireless lan interface

**Returns :**

a `YWireless` object allowing you to drive the wireless lan interface.

## YWireless.FirstWireless() yFirstWireless()yFirstWireless() YWireless

Starts the enumeration of wireless lan interfaces currently accessible.

| | |
|---|---|
| js | function **yFirstWireless**( ) |
| nodejs | function **FirstWireless**( ) |
| php | function **yFirstWireless**( ) |
| cpp | YWireless* **yFirstWireless**( ) |
| m | YWireless* **yFirstWireless**( ) |
| pas | function **yFirstWireless**( ): TYWireless |
| vb | function **yFirstWireless**( ) As YWireless |
| cs | YWireless **FirstWireless**( ) |
| java | YWireless **FirstWireless**( ) |
| py | def **FirstWireless**( ) |

Use the method `YWireless.nextWireless()` to iterate on next wireless lan interfaces.

**Returns :**

a pointer to a `YWireless` object, corresponding to the first wireless lan interface currently online, or a `null` pointer if there are none.

## wireless→adhocNetwork()[wireless adhocNetwork: ] YWireless

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

| | |
|---|---|
| js | function **adhocNetwork**( **ssid**, **securityKey**) |
| nodejs | function **adhocNetwork**( **ssid**, **securityKey**) |
| php | function **adhocNetwork**( **$ssid**, **$securityKey**) |
| cpp | int **adhocNetwork**( string **ssid**, string **securityKey**) |
| m | -(int) **adhocNetwork** : (NSString*) **ssid**<br>: (NSString*) **securityKey** |
| pas | function **adhocNetwork**( **ssid**: string, **securityKey**: string): integer |
| vb | function **adhocNetwork**( ByVal **ssid** As String,<br>ByVal **securityKey** As String) As Integer |
| cs | int **adhocNetwork**( string **ssid**, string **securityKey**) |
| java | int **adhocNetwork**( String **ssid**, String **securityKey**) |
| py | def **adhocNetwork**( **ssid**, **securityKey**) |
| cmd | YWireless **target adhocNetwork ssid securityKey** |

If a security key is specified, the network is protected by WEP128, since WPA is not standardized for ad-hoc networks. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

| | |
|---|---|
| **ssid** | the name of the network to connect to |
| **securityKey** | the network key, as a character string |

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wireless→describe()[wireless describe] YWireless

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **describe**( ) |
| nodejs | function **describe**( ) |
| php | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | function **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | def **describe**( ) |

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the wireless lan interface (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

## wireless→get_advertisedValue() wireless→advertisedValue()[wireless advertisedValue] YWireless

Returns the current value of the wireless lan interface (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| nodejs | function **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | function **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| py | def **get_advertisedValue**( ) |
| cmd | YWireless **target** **get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the wireless lan interface (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## wireless→get_channel() wireless→channel()[wireless channel] YWireless

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

| | |
|---|---|
| js | function **get_channel**( ) |
| nodejs | function **get_channel**( ) |
| php | function **get_channel**( ) |
| cpp | int **get_channel**( ) |
| m | -(int) channel |
| pas | function **get_channel**( ): LongInt |
| vb | function **get_channel**( ) As Integer |
| cs | int **get_channel**( ) |
| java | int **get_channel**( ) |
| py | def **get_channel**( ) |
| cmd | YWireless **target** **get_channel** |

**Returns :**
an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns `Y_CHANNEL_INVALID`.

## wireless→get_detectedWlans() wireless→detectedWlans()[wireless detectedWlans] YWireless

Returns a list of YWlanRecord objects that describe detected Wireless networks.

| | |
|---|---|
| js | function **get_detectedWlans**( ) |
| nodejs | function **get_detectedWlans**( ) |
| php | function **get_detectedWlans**( ) |
| cpp | vector<YWlanRecord> **get_detectedWlans**( ) |
| m | -(NSMutableArray*) detectedWlans |
| pas | function **get_detectedWlans**( ): TYWlanRecordArray |
| vb | function **get_detectedWlans**( ) As List |
| cs | List<YWlanRecord> **get_detectedWlans**( ) |
| java | ArrayList<YWlanRecord> **get_detectedWlans**( ) |
| py | def **get_detectedWlans**( ) |
| cmd | YWireless **target** **get_detectedWlans** |

This list is not updated when the module is already connected to an acces point (infrastructure mode). To force an update of this list, `adhocNetwork()` must be called to disconnect the module from the current network. The returned list must be unallocated by the caller.

**Returns :**

a list of `YWlanRecord` objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.

## wireless→get_errorMessage() wireless→errorMessage()[wireless errorMessage] YWireless

Returns the error message of the latest error with the wireless lan interface.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| nodejs | function **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | function **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | def **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using the wireless lan interface object

## wireless→get_errorType() wireless→errorType() YWireless

Returns the numerical error code of the latest error with the wireless lan interface.

| | |
|---|---|
| js | function **get_errorType**( ) |
| nodejs | function **get_errorType**( ) |
| php | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| pas | function **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | def **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occured while using the wireless lan interface object

## wireless→get_friendlyName() YWireless
## wireless→friendlyName()[wireless friendlyName]

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| nodejs | function **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | def **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the wireless lan interface if they are defined, otherwise the serial number of the module and the hardware identifier of the wireless lan interface (for exemple: `MyCustomName.relay1`)

**Returns :**
a string that uniquely identifies the wireless lan interface using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

## wireless→get_functionDescriptor() wireless→functionDescriptor()[wireless functionDescriptor] YWireless

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| nodejs | function **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | function **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | def **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

## wireless→get_functionId() wireless→functionId()[wireless functionId] YWireless

Returns the hardware identifier of the wireless lan interface, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| nodejs | function **get_functionId**( ) |
| php | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | def **get_functionId**( ) |

For example `relay1`

**Returns :**

a string that identifies the wireless lan interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## wireless→get_hardwareId() wireless→hardwareId()[wireless hardwareId] YWireless

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| nodejs | function **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | def **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wireless lan interface. (for example `RELAYLO1-123456.relay1`)

**Returns :**

a string that uniquely identifies the wireless lan interface (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

## wireless→get_linkQuality() wireless→linkQuality()[wireless linkQuality] YWireless

Returns the link quality, expressed in percent.

| | |
|---|---|
| js | function **get_linkQuality**( ) |
| nodejs | function **get_linkQuality**( ) |
| php | function **get_linkQuality**( ) |
| cpp | int **get_linkQuality**( ) |
| m | -(int) linkQuality |
| pas | function **get_linkQuality**( ): LongInt |
| vb | function **get_linkQuality**( ) As Integer |
| cs | int **get_linkQuality**( ) |
| java | int **get_linkQuality**( ) |
| py | def **get_linkQuality**( ) |
| cmd | YWireless **target** **get_linkQuality** |

**Returns :**
an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns `Y_LINKQUALITY_INVALID`.

## wireless→get_logicalName() wireless→logicalName()[wireless logicalName] YWireless

Returns the logical name of the wireless lan interface.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| nodejs | function **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | function **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| py | def **get_logicalName**( ) |
| cmd | YWireless **target** **get_logicalName** |

**Returns :**

a string corresponding to the logical name of the wireless lan interface. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## wireless→get_message() wireless→message()[wireless message] YWireless

Returns the latest status message from the wireless interface.

| | |
|---|---|
| js | function **get_message**( ) |
| nodejs | function **get_message**( ) |
| php | function **get_message**( ) |
| cpp | string **get_message**( ) |
| m | -(NSString*) message |
| pas | function **get_message**( ): string |
| vb | function **get_message**( ) As String |
| cs | string **get_message**( ) |
| java | String **get_message**( ) |
| py | def **get_message**( ) |
| cmd | YWireless **target** **get_message** |

**Returns :**
a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns `Y_MESSAGE_INVALID`.

## wireless→get_module() wireless→module()[wireless module] YWireless

Gets the `YModule` object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| nodejs | function **get_module**( ) |
| php | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | function **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| java | YModule **get_module**( ) |
| py | def **get_module**( ) |

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**
an instance of `YModule`

## wireless→get_module_async() wireless→module_async() **YWireless**

Gets the YModule object for the device on which the function is located (asynchronous version).

`js` function **get_module_async**( **callback**, **context**)
`nodejs` function **get_module_async**( **callback**, **context**)

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wireless→get_security() wireless→security()[wireless security] YWireless

Returns the security algorithm used by the selected wireless network.

| | |
|---|---|
| js | function **get_security**( ) |
| nodejs | function **get_security**( ) |
| php | function **get_security**( ) |
| cpp | Y_SECURITY_enum **get_security**( ) |
| m | -(Y_SECURITY_enum) security |
| pas | function **get_security**( ): Integer |
| vb | function **get_security**( ) As Integer |
| cs | int **get_security**( ) |
| java | int **get_security**( ) |
| py | def **get_security**( ) |
| cmd | YWireless **target** **get_security** |

**Returns :**

a value among `Y_SECURITY_UNKNOWN`, `Y_SECURITY_OPEN`, `Y_SECURITY_WEP`, `Y_SECURITY_WPA` and `Y_SECURITY_WPA2` corresponding to the security algorithm used by the selected wireless network

On failure, throws an exception or returns `Y_SECURITY_INVALID`.

## wireless→get_ssid() wireless→ssid()[wireless ssid] YWireless

Returns the wireless network name (SSID).

| | |
|---|---|
| js | function **get_ssid**( ) |
| nodejs | function **get_ssid**( ) |
| php | function **get_ssid**( ) |
| cpp | string **get_ssid**( ) |
| m | -(NSString*) ssid |
| pas | function **get_ssid**( ): string |
| vb | function **get_ssid**( ) As String |
| cs | string **get_ssid**( ) |
| java | String **get_ssid**( ) |
| py | def **get_ssid**( ) |
| cmd | YWireless **target** **get_ssid** |

**Returns :**
a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns `Y_SSID_INVALID`.

## wireless→get_userData() wireless→userData()[wireless userData] **YWireless**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| `js` | function **get_userData**( ) |
| `nodejs` | function **get_userData**( ) |
| `php` | function **get_userData**( ) |
| `cpp` | void * **get_userData**( ) |
| `m` | -(void*) userData |
| `pas` | function **get_userData**( ): Tobject |
| `vb` | function **get_userData**( ) As Object |
| `cs` | object **get_userData**( ) |
| `java` | Object **get_userData**( ) |
| `py` | def **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## wireless→isOnline()[wireless isOnline] YWireless

Checks if the wireless lan interface is currently reachable, without raising any error.

| | |
|---|---|
| js | function **isOnline**( ) |
| nodejs | function **isOnline**( ) |
| php | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | function **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | def **isOnline**( ) |

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

**Returns :**

`true` if the wireless lan interface can be reached, and `false` otherwise

## wireless→isOnline_async() YWireless

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

```
js      function isOnline_async( callback, context)
nodejs  function isOnline_async( callback, context)
```

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wireless→joinNetwork()[wireless joinNetwork: ] YWireless

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

| | |
|---|---|
| js | function **joinNetwork**( **ssid**, **securityKey**) |
| nodejs | function **joinNetwork**( **ssid**, **securityKey**) |
| php | function **joinNetwork**( **$ssid**, **$securityKey**) |
| cpp | int **joinNetwork**( string **ssid**, string **securityKey**) |
| m | -(int) **joinNetwork** : (NSString*) **ssid**<br>: (NSString*) **securityKey** |
| pas | function **joinNetwork**( **ssid**: string, **securityKey**: string): integer |
| vb | function **joinNetwork**( ByVal **ssid** As String,<br>ByVal **securityKey** As String) As Integer |
| cs | int **joinNetwork**( string **ssid**, string **securityKey**) |
| java | int **joinNetwork**( String **ssid**, String **securityKey**) |
| py | def **joinNetwork**( **ssid**, **securityKey**) |
| cmd | YWireless **target** **joinNetwork** **ssid** **securityKey** |

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

**ssid** the name of the network to connect to

**securityKey** the network key, as a character string

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## wireless→load()[wireless load: ] YWireless

Preloads the wireless lan interface cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| nodejs | function **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (int) **msValidity** |
| pas | function **load**( **msValidity**: integer): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Integer) As YRETCODE |
| cs | YRETCODE **load**( int **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | def **load**( **msValidity**) |

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## wireless→load_async() YWireless

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## wireless→nextWireless()[wireless nextWireless] YWireless

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

| | |
|---|---|
| js | function **nextWireless**( ) |
| nodejs | function **nextWireless**( ) |
| php | function **nextWireless**( ) |
| cpp | YWireless * **nextWireless**( ) |
| m | -(YWireless*) **nextWireless** |
| pas | function **nextWireless**( ): TYWireless |
| vb | function **nextWireless**( ) As YWireless |
| cs | YWireless **nextWireless**( ) |
| java | YWireless **nextWireless**( ) |
| py | def **nextWireless**( ) |

**Returns :**

a pointer to a `YWireless` object, corresponding to a wireless lan interface currently online, or a `null` pointer if there are no more wireless lan interfaces to enumerate.

## wireless→registerValueCallback()[wireless registerValueCallback: ] YWireless

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| nodejs | function **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| cpp | int **registerValueCallback**( YWirelessValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YWirelessValueCallback) **callback** |
| pas | function **registerValueCallback**( **callback**: TYWirelessValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| py | def **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## wireless→set_logicalName() wireless→setLogicalName()[wireless setLogicalName: ] YWireless

Changes the logical name of the wireless lan interface.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| nodejs | function **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| cpp | int **set_logicalName**( const string& **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | function **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| py | def **set_logicalName**( **newval**) |
| cmd | YWireless **target** **set_logicalName** **newval** |

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the wireless lan interface.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

## wireless→set_userData() wireless→setUserData()[wireless setUserData: ] YWireless

Stores a user context provided as argument in the userData attribute of the function.

| | |
|---|---|
| js | function **set_userData**( **data**) |
| nodejs | function **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| cpp | void **set_userData**( void* **data**) |
| m | -(void) setUserData : (void*) **data** |
| pas | procedure **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | def **set_userData**( **data**) |

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## wireless→wait_async() YWireless

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js      function wait_async( callback, context)
nodejs  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

# F



# G

# H



# I

# J



# L

# M



# N



# O

# P



# Q



# R



# S

# T



# U



# V



# W



# Y