Java API Reference

Table of contents

1. Introduction

This manual is intended to be used as a reference for Yoctopuce Java 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 the Yocto-Demo with Java

Java is an object oriented language created by Sun Microsystem. Beside being free, its main strength is its portability. Unfortunately, this portability has an excruciating price. In Java, hardware abstraction is so high that it is almost impossible to work directly with the hardware. Therefore, the Yoctopuce API does not support native mode in regular Java. The Java API needs a Virtual Hub to communicate with Yoctopuce devices.

2.1. Getting ready

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

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

The library is available as source files as well as a *jar* file. Decompress the library files in a folder of your choice, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface.

2.2. Control of the Led function

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

```
[...]

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("127.0.0.1");
led = YLed.FindLed("YCTOPOC1-123456.led");

// Hot-plug is easy: just check that the device is online
if (led.isOnline())
   { //Use led.set_power()
     ...
   }
```

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

```
[...]
```

Let us look at these lines in more details.

YAPI.RegisterHub

The `yAPI.RegisterHub` function initializes the Yoctopuce API and indicates where the modules should be looked for. The parameter is the address of the Virtual Hub able to see the devices. If the initialization does not succeed, an exception is thrown.

YLed.FindLed

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

```
led = YLed.FindLed("YCTOPOC1-123456.led")
led = YLed.FindLed("YCTOPOC1-123456.MyFunction")
led = YLed.FindLed("MyModule.led")
led = YLed.FindLed("MyModule.MyFunction")
led = YLed.FindLed("MyFunction")
```

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

isOnline

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

set_power

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

A real example

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

In this example, you will recognize the functions explained above, but this time used with all the side materials needed to make it work nicely as a small demo.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
import com.yoctopuce.YoctoAPI.*;

/**
 *
 * @author yocto
 */
public class Demo {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)   {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
```

```
            System.exit(1);
        }

        YLed led;
        if (args.length > 0) {
            led = YLed.FindLed(args[0]);
        } else {
            led = YLed.FirstLed();
            if (led == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        }

        try {
            System.out.println("Switch led ON");
            led.set_power(YLed.POWER_ON);
            YAPI.Sleep(1000);
            System.out.println("Switch led OFF");
            led.set_power(YLed.POWER_OFF);
        } catch (YAPI_Exception ex) {
            System.out.println("Module "+led.describe()+" not connected (check
identification and USB cable)");
        }

        YAPI.FreeAPI();
    }
}
```

2.3. Control of the module part

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

```
import com.yoctopuce.YoctoAPI.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }
        System.out.println("usage: demo [serial or logical name] [ON/OFF]");

        YModule module;
        if (args.length == 0) {
            module = YModule.FirstModule();
            if (module == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        } else {
            module = YModule.FindModule(args[0]);  // use serial or logical name
        }

        try {
            if (args.length > 1) {
                if (args[1].equalsIgnoreCase("ON")) {
                    module.setBeacon(YModule.BEACON_ON);
                } else {
                    module.setBeacon(YModule.BEACON_OFF);
                }
            }
```

```
            System.out.println("serial:       " + module.get_serialNumber());
            System.out.println("logical name: " + module.get_logicalName());
            System.out.println("luminosity:   " + module.get_luminosity());
            if (module.get_beacon() == YModule.BEACON_ON) {
                System.out.println("beacon:       ON");
            } else {
                System.out.println("beacon:       OFF");
            }
            System.out.println("upTime:       " + module.get_upTime() / 1000 + " sec");
            System.out.println("USB current:  " + module.get_usbCurrent() + " mA");
            System.out.println("logs:\n" + module.get_lastLogs());
        } catch (YAPI_Exception ex) {
            System.out.println(args[1] + " not connected (check identification and USB
cable)");
        }
        YAPI.FreeAPI();
    }
}
```

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_xxxx()`, and properties which are not read-only can be modified with the help of the `YModule.set_xxx()` method. For more details regarding the used functions, refer to the API chapters.

Changing the module settings

When you want to modify the settings of a module, you only need to call the corresponding `YModule.set_xxx()` function. However, this modification is performed only in the random access memory (RAM) of the module: if the module is restarted, the modifications are lost. To memorize them persistently, it is necessary to ask the module to save its current configuration in its permanent memory. To do so, use the `YModule.saveToFlash()` method. Inversely, it is possible to force the module to forget its current settings by using the `YModule.revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

```
import com.yoctopuce.YoctoAPI.*;

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }

        if (args.length != 2) {
            System.out.println("usage: demo <serial or logical name> <new logical name>");
            System.exit(1);
        }

        YModule m;
        String newname;

        m = YModule.FindModule(args[0]); // use serial or logical name

        try {
            newname = args[1];
            if (!YAPI.CheckLogicalName(newname))
                {
                    System.out.println("Invalid name (" + newname + ")");
                    System.exit(1);
                }

            m.set_logicalName(newname);
            m.saveToFlash(); // do not forget this

            System.out.println("Module: serial= " + m.get_serialNumber());
            System.out.println(" / name= " + m.get_logicalName());
        } catch (YAPI_Exception ex) {
            System.out.println("Module " + args[0] + "not connected (check identification
```

```
and USB cable)");
            System.out.println(ex.getMessage());
            System.exit(1);
        }

        YAPI.FreeAPI();
    }
}
```

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

Listing the modules

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

```
import com.yoctopuce.YoctoAPI.*;

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }

        System.out.println("Device list");
        YModule module = YModule.FirstModule();
        while (module != null) {
            try {
                System.out.println(module.get_serialNumber() + " (" +
module.get_productName() + ")");
            } catch (YAPI_Exception ex) {
                break;
            }
            module = module.nextModule();
        }

        YAPI.FreeAPI();
    }
}
```

2.4. Error handling

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

The simplest way to work around the problem is the one used in the short examples provided in this chapter: before accessing a module, check that it is online with the `isOnline` function, and then hope that it will stay so during the fraction of a second necessary for the following code lines to run. This method is not perfect, but it can be sufficient in some cases. You must however be aware that

you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software.

In the Java API, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash as soon as you unplug a device.

3. Reference

3.1. General functions

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

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

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

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

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

yRegisterLogFunction(**logfun**)

Registers a log callback function.

ySelectArchitecture(**arch**)

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

ySetDelegate(**object**)

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

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

Invoke the specified callback function after a given timeout.

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

Pauses the execution flow for a specified duration.

yTriggerHubDiscovery(**errmsg**)

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

yUnregisterHub(**url**)

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

yUpdateDeviceList(**errmsg**)

Triggers a (re)detection of connected Yoctopuce modules.

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

Triggers a (re)detection of connected Yoctopuce modules.

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

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

boolean **CheckLogicalName**(String **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.EnableUSBHost() YAPI
yEnableUSBHost()YAPI.EnableUSBHost()

This function is used only on Android.

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() YAPI
yFreeAPI()YAPI.FreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

void **FreeAPI**( )

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

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

Returns the version identifier for the Yoctopuce library in use.

String **GetAPIVersion**( )

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

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

Returns :
a character string describing the library version.

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

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

long **GetTickCount**( )

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

Returns :
a long integer corresponding to the millisecond counter.

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

Maintains the device-to-library communication channel.

```
int HandleEvents( )
```

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

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

Parameters :

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

Returns :

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

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

Initializes the Yoctopuce programming library explicitly.

int **InitAPI**(int **mode**)

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

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

Parameters :

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

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

Returns :

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

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

Fault-tolerant alternative to RegisterHub().

int **PreregisterHub**(String **url**)

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

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

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

void **RegisterDeviceArrivalCallback**(DeviceArrivalCallback **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() YAPI
yRegisterDeviceRemovalCallback()
YAPI.RegisterDeviceRemovalCallback()

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

void **RegisterDeviceRemovalCallback**(DeviceRemovalCallback **removalCallback**)

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

Parameters :

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

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

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

int **RegisterHub**(String **url**)

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() YAPI
yRegisterHubDiscoveryCallback()
YAPI.RegisterHubDiscoveryCallback()

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

void **RegisterHubDiscoveryCallback**(HubDiscoveryCallback **hubDiscoveryCallback**)

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

Parameters :

hubDiscoveryCallback a procedure taking two string parameter, or null

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

Registers a log callback function.

void **RegisterLogFunction**(LogCallback **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.Sleep() YAPI
ySleep()YAPI.Sleep()

Pauses the execution flow for a specified duration.

int **Sleep**(long **ms_duration**)

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() YAPI
yTriggerHubDiscovery()
YAPI.TriggerHubDiscovery()

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.

int **TriggerHubDiscovery**( )

Parameters :

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

Returns :

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

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

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

```
void UnregisterHub( String url)
```

Parameters :

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

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

Triggers a (re)detection of connected Yoctopuce modules.

int **UpdateDeviceList**( )

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

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

Parameters :

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

Returns :

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

3.2. Accelerometer function interface

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

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

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

Global functions
yFindAccelerometer(**func**) Retrieves an accelerometer for a given identifier.
yFirstAccelerometer() Starts the enumeration of accelerometers currently accessible.
`YAccelerometer` methods
accelerometer→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
accelerometer→describe() Returns a short text that describes unambiguously the instance of the accelerometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
accelerometer→get_advertisedValue() Returns the current value of the accelerometer (no more than 6 characters).
accelerometer→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
accelerometer→get_currentValue() Returns the current value of the acceleration.
accelerometer→get_errorMessage() Returns the error message of the latest error with the accelerometer.
accelerometer→get_errorType() Returns the numerical error code of the latest error with the accelerometer.
accelerometer→get_friendlyName() Returns a global identifier of the accelerometer in the format `MODULE_NAME.FUNCTION_NAME`.
accelerometer→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
accelerometer→get_functionId() Returns the hardware identifier of the accelerometer, without reference to the module.
accelerometer→get_hardwareId() Returns the unique hardware identifier of the accelerometer in the form `SERIAL.FUNCTIONID`.

accelerometer→get_highestValue()

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

accelerometer→get_logFrequency()

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

accelerometer→get_logicalName()

Returns the logical name of the accelerometer.

accelerometer→get_lowestValue()

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

accelerometer→get_module()

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

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

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

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

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

accelerometer→get_reportFrequency()

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

accelerometer→get_resolution()

Returns the resolution of the measured values.

accelerometer→get_unit()

Returns the measuring unit for the acceleration.

accelerometer→get_userData()

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

accelerometer→get_xValue()

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

accelerometer→get_yValue()

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

accelerometer→get_zValue()

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

accelerometer→isOnline()

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

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

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

accelerometer→load(**msValidity**)

Preloads the accelerometer cache with a specified validity duration.

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

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

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

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

accelerometer→nextAccelerometer()

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

accelerometer→registerTimedReportCallback(**callback**)

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

accelerometer→registerValueCallback(**callback**)

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

accelerometer→set_highestValue(newval)

Changes the recorded maximal value observed.

accelerometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

accelerometer→set_logicalName(newval)

Changes the logical name of the accelerometer.

accelerometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

accelerometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

accelerometer→set_resolution(newval)

Changes the resolution of the measured physical values.

accelerometer→set_userData(data)

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

accelerometer→wait_async(callback, context)

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

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

Retrieves an accelerometer for a given identifier.

YAccelerometer **FindAccelerometer**(String **func**)

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() YAccelerometer
yFirstAccelerometer()
YAccelerometer.FirstAccelerometer()

Starts the enumeration of accelerometers currently accessible.

YAccelerometer **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() YAccelerometer
accelerometer.calibrateFromPoints()

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

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                                                ArrayList<Double> 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() YAccelerometer
accelerometer.describe()

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

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

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

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

String **get_advertisedValue**( )

Returns :

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

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

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

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

Returns the current value of the acceleration.

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

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

String **get_errorMessage**( )

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

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

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

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

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

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

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

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

String **get_functionDescriptor**( )

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

Returns :

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

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

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

String **get_functionId**( )

For example `relay1`

Returns :

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

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

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

String **get_hardwareId**( )

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

Returns :

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

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

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

double **get_highestValue**( )

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

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

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

String **get_logFrequency**( )

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

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

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

Returns the logical name of the accelerometer.

String **get_logicalName**( )

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

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

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

double **get_lowestValue**( )

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

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

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

YModule **get_module**( )

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

Returns :
an instance of YModule

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

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

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

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

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

Parameters :

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

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

Returns :

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

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

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

String **get_reportFrequency**( )

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

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

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

Returns the resolution of the measured values.

double **get_resolution**( )

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

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

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

Returns the measuring unit for the acceleration.

String **get_unit**( )

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

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

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

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

Object **get_userData**( )

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

Returns :
the object stored previously by the caller.

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

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

double **get_xValue**( )

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

On failure, throws an exception or returns Y_XVALUE_INVALID.

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

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

double **get_yValue**( )

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

On failure, throws an exception or returns Y_YVALUE_INVALID.

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

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

double **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() YAccelerometer
accelerometer.isOnline()

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

boolean **isOnline**( )

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

Returns :

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

accelerometer→load()accelerometer.load() YAccelerometer

Preloads the accelerometer cache with a specified validity duration.

int **load**(long **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() YAccelerometer
accelerometer.loadCalibrationPoints()

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

int **loadCalibrationPoints**(ArrayList<Double> **rawValues**,
ArrayList<Double> **refValues**)

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→nextAccelerometer() YAccelerometer
accelerometer.nextAccelerometer()

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

YAccelerometer **nextAccelerometer**( )

Returns :

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

accelerometer→registerTimedReportCallback() YAccelerometer
accelerometer.registerTimedReportCallback( )

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

int **registerTimedReportCallback**(TimedReportCallback **callback**)

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

Parameters :

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

accelerometer→registerValueCallback() YAccelerometer
accelerometer.registerValueCallback()

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

int **registerValueCallback**(UpdateCallback **callback**)

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

Parameters :

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

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

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the accelerometer.

int **set_logicalName**(String **newval**)

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

Parameters :

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

Returns :

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

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

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

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

void **set_userData**(Object **data**)

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

Parameters :

data any kind of object to be stored

3.3. AnButton function interface

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

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

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

Global functions
yFindAnButton(**func**) Retrieves an analog input for a given identifier.
yFirstAnButton() Starts the enumeration of analog inputs currently accessible.
YAnButton methods
anbutton→describe() Returns a short text that describes unambiguously the instance of the analog input in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
anbutton→get_advertisedValue() Returns the current value of the analog input (no more than 6 characters).
anbutton→get_analogCalibration() Tells if a calibration process is currently ongoing.
anbutton→get_calibratedValue() Returns the current calibrated input value (between 0 and 1000, included).
anbutton→get_calibrationMax() Returns the maximal value measured during the calibration (between 0 and 4095, included).
anbutton→get_calibrationMin() Returns the minimal value measured during the calibration (between 0 and 4095, included).
anbutton→get_errorMessage() Returns the error message of the latest error with the analog input.
anbutton→get_errorType() Returns the numerical error code of the latest error with the analog input.
anbutton→get_friendlyName() Returns a global identifier of the analog input in the format `MODULE_NAME.FUNCTION_NAME`.
anbutton→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

anbutton→get_functionId()

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

anbutton→get_hardwareId()

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

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

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

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

anbutton→get_pulseCounter()

Returns the pulse counter value

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms)

anbutton→get_rawValue()

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

anbutton→get_sensitivity()

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

anbutton→get_userData()

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

anbutton→isOnline()

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

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

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

anbutton→load(**msValidity**)

Preloads the analog input cache with a specified validity duration.

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

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

anbutton→nextAnButton()

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

anbutton→registerValueCallback(**callback**)

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

anbutton→resetCounter()

Returns the pulse counter value as well as his timer

anbutton→set_analogCalibration(**newval**)

Starts or stops the calibration process.

anbutton→set_calibrationMax(**newval**)

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

anbutton→set_calibrationMin(newval)

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

anbutton→set_logicalName(newval)

Changes the logical name of the analog input.

anbutton→set_sensitivity(newval)

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

anbutton→set_userData(data)

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

anbutton→wait_async(callback, context)

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

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

Retrieves an analog input for a given identifier.

YAnButton **FindAnButton**(String **func**)

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the analog input

Returns :

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

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

Starts the enumeration of analog inputs currently accessible.

YAnButton **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.

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 analog input (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

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

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

String **get_advertisedValue**( )

Returns :

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

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

Tells if a calibration process is currently ongoing.

int **get_analogCalibration**( )

Returns :
either Y_ANALOGCALIBRATION_OFF or Y_ANALOGCALIBRATION_ON

On failure, throws an exception or returns Y_ANALOGCALIBRATION_INVALID.

anbutton→get_calibratedValue() YAnButton
anbutton→calibratedValue()
anbutton.get_calibratedValue()

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

int **get_calibratedValue**( )

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

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

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

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

int **get_calibrationMax**( )

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

On failure, throws an exception or returns Y_CALIBRATIONMAX_INVALID.

anbutton→get_calibrationMin() YAnButton
anbutton→calibrationMin()
anbutton.get_calibrationMin()

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

int **get_calibrationMin**( )

Returns :

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

On failure, throws an exception or returns Y_CALIBRATIONMIN_INVALID.

anbutton→get_errorMessage() YAnButton
anbutton→errorMessage()
anbutton.get_errorMessage()

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

String **get_errorMessage**( )

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

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

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

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

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

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

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

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

String **get_functionDescriptor**( )

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

Returns :

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

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

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

String **get_functionId**( )

For example `relay1`

Returns :

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

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

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

String **get_hardwareId**( )

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

Returns :

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

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

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

int **get_isPressed**( )

Returns :

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

On failure, throws an exception or returns Y_ISPRESSED_INVALID.

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

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

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

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

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

Returns the logical name of the analog input.

String **get_logicalName**( )

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

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

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

YModule **get_module**( )

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

Returns :
an instance of YModule

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

Returns the pulse counter value

long **get_pulseCounter**( )

Returns :
an integer corresponding to the pulse counter value

On failure, throws an exception or returns Y_PULSECOUNTER_INVALID.

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

Returns the timer of the pulses counter (ms)

long **get_pulseTimer**( )

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

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

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

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

int **get_rawValue**( )

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

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

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

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

int **get_sensitivity**( )

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

On failure, throws an exception or returns Y_SENSITIVITY_INVALID.

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

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

Object **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.

boolean **isOnline**( )

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

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

anbutton→load()anbutton.load() YAnButton

Preloads the analog input cache with a specified validity duration.

int **load**(long **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→nextAnButton() YAnButton
anbutton.nextAnButton()

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

YAnButton **nextAnButton**( )

Returns :

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

anbutton→registerValueCallback() YAnButton
anbutton.registerValueCallback()

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

int **registerValueCallback**(UpdateCallback **callback**)

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

Parameters :

callback the callback 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() YAnButton
anbutton.resetCounter()

Returns the pulse counter value as well as his timer

int **resetCounter**( )

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→set_analogCalibration() YAnButton
anbutton→setAnalogCalibration()
anbutton.set_analogCalibration()

Starts or stops the calibration process.

int **set_analogCalibration**(int **newval**)

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

Parameters :

newval either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton→set_calibrationMax() YAnButton
anbutton→setCalibrationMax()
anbutton.set_calibrationMax()

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

int **set_calibrationMax**(int **newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton→set_calibrationMin() YAnButton
anbutton→setCalibrationMin()
anbutton.set_calibrationMin()

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

int **set_calibrationMin**(int **newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton→set_logicalName() YAnButton
anbutton→setLogicalName()
anbutton.set_logicalName()

Changes the logical name of the analog input.

int **set_logicalName**(String **newval**)

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

Parameters :

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

Returns :

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

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

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

int **set_sensitivity**(int **newval**)

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

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

void **set_userData**(Object **data**)

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

Parameters :

data any kind of object to be stored

3.4. CarbonDioxide function interface

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

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

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

Global functions
yFindCarbonDioxide(**func**) Retrieves a CO2 sensor for a given identifier.
yFirstCarbonDioxide() Starts the enumeration of CO2 sensors currently accessible.
`YCarbonDioxide` methods
carbondioxide→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
carbondioxide→describe() Returns a short text that describes unambiguously the instance of the CO2 sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
carbondioxide→get_advertisedValue() Returns the current value of the CO2 sensor (no more than 6 characters).
carbondioxide→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
carbondioxide→get_currentValue() Returns the current value of the CO2 concentration.
carbondioxide→get_errorMessage() Returns the error message of the latest error with the CO2 sensor.
carbondioxide→get_errorType() Returns the numerical error code of the latest error with the CO2 sensor.
carbondioxide→get_friendlyName() Returns a global identifier of the CO2 sensor in the format `MODULE_NAME.FUNCTION_NAME`.
carbondioxide→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
carbondioxide→get_functionId() Returns the hardware identifier of the CO2 sensor, without reference to the module.
carbondioxide→get_hardwareId() Returns the unique hardware identifier of the CO2 sensor in the form `SERIAL.FUNCTIONID`.

carbondioxide→get_highestValue()

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

carbondioxide→get_logFrequency()

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

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

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

carbondioxide→get_module()

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

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

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

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

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

carbondioxide→get_reportFrequency()

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_unit()

Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

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

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

carbondioxide→load(**msValidity**)

Preloads the CO2 sensor cache with a specified validity duration.

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

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

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

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

carbondioxide→nextCarbonDioxide()

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

carbondioxide→registerTimedReportCallback(**callback**)

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

carbondioxide→registerValueCallback(**callback**)

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

carbondioxide→set_highestValue(**newval**)

Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(**newval**)

Changes the logical name of the CO2 sensor.

carbondioxide→set_lowestValue(newval)

Changes the recorded minimal value observed.

carbondioxide→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

carbondioxide→set_resolution(newval)

Changes the resolution of the measured physical values.

carbondioxide→set_userData(data)

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

carbondioxide→wait_async(callback, context)

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

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

Retrieves a CO2 sensor for a given identifier.

YCarbonDioxide **FindCarbonDioxide**(String **func**)

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() YCarbonDioxide yFirstCarbonDioxide() `YCarbonDioxide.FirstCarbonDioxide()`

Starts the enumeration of CO2 sensors currently accessible.

YCarbonDioxide **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() YCarbonDioxide
carbondioxide.calibrateFromPoints()

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

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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() YCarbonDioxide
carbondioxide.describe()

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

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

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

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

String **get_advertisedValue**( )

Returns :

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

carbondioxide→get_currentRawValue() YCarbonDioxide
carbondioxide→currentRawValue()
carbondioxide.get_currentRawValue()

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

double **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() YCarbonDioxide
carbondioxide→currentValue()
carbondioxide.get_currentValue()

Returns the current value of the CO2 concentration.

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

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

String **get_errorMessage**( )

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

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

carbondioxide→get_errorType() YCarbonDioxide
carbondioxide→errorType()
carbondioxide.get_errorType()

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

```
int 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() YCarbonDioxide
carbondioxide→friendlyName()
carbondioxide.get_friendlyName()

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

String **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() YCarbonDioxide
carbondioxide→functionDescriptor()
carbondioxide.get_functionDescriptor()

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

String **get_functionDescriptor**( )

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

Returns :

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

carbondioxide→get_functionId() YCarbonDioxide
carbondioxide→functionId()
carbondioxide.get_functionId()

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

String **get_functionId**( )

For example `relay1`

Returns :

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

carbondioxide→get_hardwareId() YCarbonDioxide
carbondioxide→hardwareId()
carbondioxide.get_hardwareId()

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

String **get_hardwareId**( )

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

Returns :

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

carbondioxide→get_highestValue() YCarbonDioxide
carbondioxide→highestValue()
carbondioxide.get_highestValue()

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

double **get_highestValue**( )

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

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

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

String **get_logFrequency**( )

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

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

carbondioxide→get_logicalName() YCarbonDioxide
carbondioxide→logicalName()
carbondioxide.get_logicalName()

Returns the logical name of the CO2 sensor.

String **get_logicalName**( )

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

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

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

double **get_lowestValue**( )

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

carbondioxide→get_module() YCarbonDioxide
carbondioxide→module()
carbondioxide.get_module()

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

YModule **get_module**( )

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

Returns :
an instance of YModule

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

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

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

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

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

Parameters :

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

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

Returns :

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

carbondioxide→get_reportFrequency() YCarbonDioxide
carbondioxide→reportFrequency()
`carbondioxide.get_reportFrequency()`

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

String **get_reportFrequency**( )

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

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

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

Returns the resolution of the measured values.

double **get_resolution**( )

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

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

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

Returns the measuring unit for the CO2 concentration.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the CO2 concentration

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

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

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

Object **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() YCarbonDioxide
carbondioxide.isOnline()

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

boolean **isOnline**( )

If there is a cached value for the CO2 sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the CO2 sensor.

Returns :
`true` if the CO2 sensor can be reached, and `false` otherwise

carbondioxide→load()`carbondioxide.load()` YCarbonDioxide

Preloads the CO2 sensor cache with a specified validity duration.

int **load**(long **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() YCarbonDioxide
carbondioxide.loadCalibrationPoints()

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

int **loadCalibrationPoints**(ArrayList<Double> **rawValues**,
ArrayList<Double> **refValues**)

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→nextCarbonDioxide() YCarbonDioxide
carbondioxide.nextCarbonDioxide()

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

YCarbonDioxide **nextCarbonDioxide**( )

Returns :

a pointer to a `YCarbonDioxide` object, corresponding to a CO2 sensor currently online, or a `null` pointer if there are no more CO2 sensors to enumerate.

carbondioxide→registerTimedReportCallback() YCarbonDioxide
carbondioxide.registerTimedReportCallback( )

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

int **registerTimedReportCallback**(TimedReportCallback **callback**)

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

Parameters :

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

carbondioxide→registerValueCallback() YCarbonDioxide
carbondioxide.registerValueCallback()

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

int **registerValueCallback**(UpdateCallback **callback**)

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

Parameters :

callback the callback 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() YCarbonDioxide
carbondioxide→setHighestValue()
carbondioxide.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logFrequency() YCarbonDioxide
carbondioxide→setLogFrequency()
carbondioxide.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logicalName() YCarbonDioxide
carbondioxide→setLogicalName()
carbondioxide.set_logicalName()

Changes the logical name of the CO2 sensor.

int **set_logicalName**(String **newval**)

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

Parameters :

newval a string corresponding to the logical name of the CO2 sensor.

Returns :

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

carbondioxide→set_lowestValue() YCarbonDioxide
carbondioxide→setLowestValue()
carbondioxide.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_reportFrequency() YCarbonDioxide
carbondioxide→setReportFrequency()
carbondioxide.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_resolution() YCarbonDioxide
carbondioxide→setResolution()
carbondioxide.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_userData() YCarbonDioxide
carbondioxide→setUserData()
carbondioxide.set_userData()

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

void **set_userData**(Object **data**)

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

Parameters :

data any kind of object to be stored

3.5. ColorLed function interface

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

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

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

Global functions
yFindColorLed(**func**) Retrieves an RGB led for a given identifier.
yFirstColorLed() Starts the enumeration of RGB leds currently accessible.
`YColorLed` methods
colorled→describe() Returns a short text that describes unambiguously the instance of the RGB led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
colorled→get_advertisedValue() Returns the current value of the RGB led (no more than 6 characters).
colorled→get_errorMessage() Returns the error message of the latest error with the RGB led.
colorled→get_errorType() Returns the numerical error code of the latest error with the RGB led.
colorled→get_friendlyName() Returns a global identifier of the RGB led in the format `MODULE_NAME.FUNCTION_NAME`.
colorled→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
colorled→get_functionId() Returns the hardware identifier of the RGB led, without reference to the module.
colorled→get_hardwareId() Returns the unique hardware identifier of the RGB led in the form `SERIAL.FUNCTIONID`.
colorled→get_hslColor() Returns the current HSL color of the led.
colorled→get_logicalName() Returns the logical name of the RGB led.

colorled→get_module()

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

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

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

colorled→get_rgbColor()

Returns the current RGB color of the led.

colorled→get_rgbColorAtPowerOn()

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

colorled→get_userData()

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

colorled→hslMove(**hsl_target**, **ms_duration**)

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

colorled→isOnline()

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

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

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

colorled→load(**msValidity**)

Preloads the RGB led cache with a specified validity duration.

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

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

colorled→nextColorLed()

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

colorled→registerValueCallback(**callback**)

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

colorled→rgbMove(**rgb_target**, **ms_duration**)

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

colorled→set_hslColor(**newval**)

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

colorled→set_logicalName(**newval**)

Changes the logical name of the RGB led.

colorled→set_rgbColor(**newval**)

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

colorled→set_rgbColorAtPowerOn(**newval**)

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

colorled→set_userData(**data**)

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

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

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

YColorLed.FindColorLed() YColorLed yFindColorLed()YColorLed.FindColorLed()

Retrieves an RGB led for a given identifier.

YColorLed **FindColorLed**(String **func**)

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() YColorLed yFirstColorLed()YColorLed.FirstColorLed()

Starts the enumeration of RGB leds currently accessible.

```
YColorLed 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`.

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

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

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

String **get_advertisedValue**( )

Returns :

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

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

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

String **get_errorMessage**( )

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

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

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

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

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

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

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

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

String **get_functionDescriptor**( )

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

Returns :

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

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

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

String **get_functionId**( )

For example `relay1`

Returns :

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

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

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

String **get_hardwareId**( )

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

Returns :

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

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

Returns the current HSL color of the led.

int **get_hslColor**( )

Returns :
an integer corresponding to the current HSL color of the led

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

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

Returns the logical name of the RGB led.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the RGB led. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

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

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

YModule **get_module**( )

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

Returns :
an instance of YModule

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

Returns the current RGB color of the led.

int **get_rgbColor**( )

Returns :
an integer corresponding to the current RGB color of the led

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

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

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

int **get_rgbColorAtPowerOn**( )

Returns :
an integer corresponding to the configured color to be displayed when the module is turned on

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

colorled→get_userData() YColorLed
colorled→userData()`colorled.get_userData()`

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

Object **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.

int **hslMove**(int **hsl_target**, int **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.

boolean **isOnline**( )

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

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

colorled→load()`colorled.load()` YColorLed

Preloads the RGB led cache with a specified validity duration.

int **load**(long **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→nextColorLed() YColorLed
colorled.nextColorLed()

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

YColorLed **nextColorLed**( )

Returns :

a pointer to a `YColorLed` object, corresponding to an RGB led currently online, or a `null` pointer if there are no more RGB leds to enumerate.

colorled→registerValueCallback() YColorLed
colorled.registerValueCallback()

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

int **registerValueCallback**(UpdateCallback **callback**)

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

Parameters :

callback the callback 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.

int **rgbMove**(int **rgb_target**, int **ms_duration**)

Parameters :

rgb_target	desired RGB color at the end of the transition
ms_duration	duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

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

int **set_hslColor**(int **newval**)

Encoding is done as follows: 0xHHSSLL.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the logical name of the RGB led.

int **set_logicalName**(String **newval**)

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

Parameters :

newval a string corresponding to the logical name of the RGB led.

Returns :

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

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

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

int **set_rgbColor**(int **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() YColorLed
colorled→setRgbColorAtPowerOn()
colorled.set_rgbColorAtPowerOn()

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

int **set_rgbColorAtPowerOn**(int **newval**)

This color will be displayed as soon as the module is powered on. Remember to call the `saveToFlash()` method of the module if the change should be kept.

Parameters :

newval an integer corresponding to the color that the led will display by default when the module is turned on

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

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

void **set_userData**(Object **data**)

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

Parameters :

data any kind of object to be stored

3.6. Compass function interface

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

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

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

Global functions
yFindCompass(**func**)
Retrieves a compass for a given identifier.
yFirstCompass()
Starts the enumeration of compasses currently accessible.
YCompass methods
compass→calibrateFromPoints(**rawValues**, **refValues**)
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
compass→describe()
Returns a short text that describes unambiguously the instance of the compass in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
compass→get_advertisedValue()
Returns the current value of the compass (no more than 6 characters).
compass→get_currentRawValue()
Returns the uncalibrated, unrounded raw value returned by the sensor.
compass→get_currentValue()
Returns the current value of the relative bearing.
compass→get_errorMessage()
Returns the error message of the latest error with the compass.
compass→get_errorType()
Returns the numerical error code of the latest error with the compass.
compass→get_friendlyName()
Returns a global identifier of the compass in the format `MODULE_NAME.FUNCTION_NAME`.
compass→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
compass→get_functionId()
Returns the hardware identifier of the compass, without reference to the module.
compass→get_hardwareId()
Returns the unique hardware identifier of the compass in the form `SERIAL.FUNCTIONID`.

compass→get_highestValue()

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

compass→get_logFrequency()

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

compass→get_logicalName()

Returns the logical name of the compass.

compass→get_lowestValue()

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

compass→get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

compass→get_module()

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

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

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

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

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

compass→get_reportFrequency()

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

compass→get_resolution()

Returns the resolution of the measured values.

compass→get_unit()

Returns the measuring unit for the relative bearing.

compass→get_userData()

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

compass→isOnline()

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

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

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

compass→load(**msValidity**)

Preloads the compass cache with a specified validity duration.

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

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

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

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

compass→nextCompass()

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

compass→registerTimedReportCallback(**callback**)

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

compass→registerValueCallback(**callback**)

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

compass→set_highestValue(**newval**)

Changes the recorded maximal value observed.

compass→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

compass→set_logicalName(newval)

Changes the logical name of the compass.

compass→set_lowestValue(newval)

Changes the recorded minimal value observed.

compass→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

compass→set_resolution(newval)

Changes the resolution of the measured physical values.

compass→set_userData(data)

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

compass→wait_async(callback, context)

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

YCompass.FindCompass() YCompass yFindCompass()YCompass.FindCompass()

Retrieves a compass for a given identifier.

YCompass **FindCompass**(String **func**)

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() YCompass yFirstCompass()YCompass.FirstCompass()

Starts the enumeration of compasses currently accessible.

```
YCompass 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() YCompass
compass.calibrateFromPoints()

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

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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

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

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

String **get_advertisedValue**( )

Returns :

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

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

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

double **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() YCompass
compass→currentValue()
compass.get_currentValue()

Returns the current value of the relative bearing.

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

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

String **get_errorMessage**( )

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

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

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

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

int **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() YCompass
compass→friendlyName()
compass.get_friendlyName()

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

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

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

String **get_functionDescriptor**( )

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

Returns :

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

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

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

String **get_functionId**( )

For example `relay1`

Returns :

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

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

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

String **get_hardwareId**( )

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

Returns :

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

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

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

double **get_highestValue**( )

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

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

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

String **get_logFrequency**( )

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

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

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

Returns the logical name of the compass.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the compass. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

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

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

double **get_lowestValue**( )

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

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

Returns the magnetic heading, regardless of the configured bearing.

double **get_magneticHeading**( )

Returns :
a floating point number corresponding to the magnetic heading, regardless of the configured bearing

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

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

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

YModule **get_module**( )

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

Returns :
an instance of YModule

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

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

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

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

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

Parameters :

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

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

Returns :

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

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

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

String **get_reportFrequency**( )

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

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

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

Returns the resolution of the measured values.

double **get_resolution**( )

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

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

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

Returns the measuring unit for the relative bearing.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the relative bearing

On failure, throws an exception or returns Y_UNIT_INVALID.

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

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

Object **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.

boolean **isOnline**( )

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

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

compass→load()compass.load() YCompass

Preloads the compass cache with a specified validity duration.

int **load**(long **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() YCompass
compass.loadCalibrationPoints()

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

int **loadCalibrationPoints**(ArrayList<Double> **rawValues**,
ArrayList<Double> **refValues**)

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→nextCompass()compass.nextCompass() YCompass

Continues the enumeration of compasses started using yFirstCompass().

YCompass **nextCompass**( )

Returns :

a pointer to a YCompass object, corresponding to a compass currently online, or a null pointer if there are no more compasses to enumerate.

compass→registerTimedReportCallback() YCompass
compass.registerTimedReportCallback()

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

int **registerTimedReportCallback**(TimedReportCallback **callback**)

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

Parameters :

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

compass→registerValueCallback() YCompass
compass.registerValueCallback()

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

int **registerValueCallback**(UpdateCallback **callback**)

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

Parameters :

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

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

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set_logFrequency() YCompass
compass→setLogFrequency()
compass.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the compass.

int **set_logicalName**(String **newval**)

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

Parameters :

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

Returns :

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

compass→set_lowestValue() YCompass
compass→setLowestValue()
compass.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

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

void **set_userData**(Object **data**)

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

Parameters :

data any kind of object to be stored

3.7. Current function interface

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

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

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

Global functions

yFindCurrent(**func**)

Retrieves a current sensor for a given identifier.

yFirstCurrent()

Starts the enumeration of current sensors currently accessible.

`YCurrent` methods

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

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

current→describe()

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

current→get_advertisedValue()

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

current→get_currentRawValue()

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

current→get_currentValue()

Returns the current measure for the current.

current→get_errorMessage()

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

current→get_errorType()

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

current→get_friendlyName()

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

current→get_functionDescriptor()

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

current→get_functionId()

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

current→get_hardwareId()

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

current→get_highestValue()

Returns the maximal value observed for the current.

current→get_logFrequency()

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

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

Returns the minimal value observed for the current.

current→get_module()

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

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

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

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

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

current→get_reportFrequency()

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

current→get_resolution()

Returns the resolution of the measured values.

current→get_unit()

Returns the measuring unit for the current.

current→get_userData()

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

current→isOnline()

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

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

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

current→load(**msValidity**)

Preloads the current sensor cache with a specified validity duration.

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

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

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

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

current→nextCurrent()

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

current→registerTimedReportCallback(**callback**)

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

current→registerValueCallback(**callback**)

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

current→set_highestValue(**newval**)

Changes the recorded maximal value observed pour the current.

current→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

current→set_logicalName(**newval**)

Changes the logical name of the current sensor.

current→set_lowestValue(newval)

Changes the recorded minimal value observed pour the current.

current→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

current→set_resolution(newval)

Changes the resolution of the measured values.

current→set_userData(data)

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

current→wait_async(callback, context)

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

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

Retrieves a current sensor for a given identifier.

YCurrent **FindCurrent**(String **func**)

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the current sensor

Returns :

a `YCurrent` object allowing you to drive the current sensor.

YCurrent.FirstCurrent() YCurrent yFirstCurrent()YCurrent.FirstCurrent()

Starts the enumeration of current sensors currently accessible.

YCurrent **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() YCurrent
current.calibrateFromPoints()

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

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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

current→get_advertisedValue() YCurrent
current→advertisedValue()
current.get_advertisedValue()

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

String **get_advertisedValue**( )

Returns :

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

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

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

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

Returns the current measure for the current.

double **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.get_errorMessage()

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

String **get_errorMessage**( )

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

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

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

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

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

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

String **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() YCurrent
current→functionDescriptor()
current.get_functionDescriptor()

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

String **get_functionDescriptor**( )

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

Returns :

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

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

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

String **get_functionId**( )

For example `relay1`

Returns :

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

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

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

String **get_hardwareId**( )

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

Returns :

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

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

Returns the maximal value observed for the current.

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

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

String **get_logFrequency**( )

Returns :

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

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

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

Returns the logical name of the current sensor.

String **get_logicalName**( )

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

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

Returns the minimal value observed for the current.

double **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() YCurrent
current→module()`current.get_module()`

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

```
YModule get_module( )
```

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

Returns :
an instance of `YModule`

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

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

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

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

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

Parameters :

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

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

Returns :

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

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

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

String **get_reportFrequency**( )

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

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

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

Returns the resolution of the measured values.

double **get_resolution**( )

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

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

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

Returns the measuring unit for the current.

String **get_unit**( )

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

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

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

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

Object **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.

boolean **isOnline**( )

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

Returns :

`true` if the current sensor can be reached, and `false` otherwise

current→load()current.load() YCurrent

Preloads the current sensor cache with a specified validity duration.

int **load**(long **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() YCurrent
current.loadCalibrationPoints()

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

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→nextCurrent()current.nextCurrent() YCurrent

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

YCurrent **nextCurrent**( )

Returns :

a pointer to a YCurrent object, corresponding to a current sensor currently online, or a null pointer if there are no more current sensors to enumerate.

current→registerTimedReportCallback() YCurrent
current.registerTimedReportCallback()

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

int **registerTimedReportCallback**(TimedReportCallback **callback**)

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

Parameters :

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

current→registerValueCallback() YCurrent
current.registerValueCallback()

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

int **registerValueCallback**(UpdateCallback **callback**)

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

Parameters :

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

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

Changes the recorded maximal value observed pour the current.

int **set_highestValue**(double **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() YCurrent
current→setLogFrequency()
current.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the current sensor.

int **set_logicalName**(String **newval**)

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

Parameters :

newval a string corresponding to the logical name of the current sensor.

Returns :

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

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

Changes the recorded minimal value observed pour the current.

int **set_lowestValue**(double **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() YCurrent
current→setReportFrequency()
current.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the resolution of the measured values.

int **set_resolution**(double **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() YCurrent
current→setUserData()`current.set_userData()`

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

void **set_userData**(Object **data**)

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

Parameters :

data any kind of object to be stored

3.8. DataLogger function interface

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

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

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

Global functions
yFindDataLogger(**func**) Retrieves a data logger for a given identifier.
yFirstDataLogger() Starts the enumeration of data loggers currently accessible.
`YDataLogger` methods
datalogger→describe() Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
datalogger→forgetAllDataStreams() Clears the data logger memory and discards all recorded data streams.
datalogger→get_advertisedValue() Returns the current value of the data logger (no more than 6 characters).
datalogger→get_autoStart() Returns the default activation state of the data logger on power up.
datalogger→get_currentRunIndex() Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.
datalogger→get_dataSets() Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.
datalogger→get_dataStreams(**v**) Builds a list of all data streams hold by the data logger (legacy method).
datalogger→get_errorMessage() Returns the error message of the latest error with the data logger.
datalogger→get_errorType() Returns the numerical error code of the latest error with the data logger.
datalogger→get_friendlyName() Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.
datalogger→get_functionDescriptor()

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

datalogger→get_functionId()

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

datalogger→get_hardwareId()

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

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_module()

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

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

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

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_timeUTC()

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

datalogger→get_userData()

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

datalogger→isOnline()

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

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

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

datalogger→load(**msValidity**)

Preloads the data logger cache with a specified validity duration.

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

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

datalogger→nextDataLogger()

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

datalogger→registerValueCallback(**callback**)

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

datalogger→set_autoStart(**newval**)

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

datalogger→set_logicalName(**newval**)

Changes the logical name of the data logger.

datalogger→set_recording(**newval**)

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

datalogger→set_timeUTC(**newval**)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(**data**)

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

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

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

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

Retrieves a data logger for a given identifier.

YDataLogger **FindDataLogger**(String **func**)

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger() YDataLogger
yFirstDataLogger()
YDataLogger.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

YDataLogger **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.

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 data logger (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

datalogger→forgetAllDataStreams() YDataLogger
datalogger.forgetAllDataStreams()

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

int **forgetAllDataStreams**( )

This method also resets the current run index to zero.

Returns :
YAPI_SUCCESS if the call succeeds.

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

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

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

String **get_advertisedValue**( )

Returns :

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

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

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

int **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() YDataLogger
datalogger→currentRunIndex()
`datalogger.get_currentRunIndex()`

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

int **get_currentRunIndex**( )

Returns :

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

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

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

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

```
ArrayList<YDataSet> get_dataSets( )
```

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

Returns :
a list of YDataSet object.

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

datalogger→get_dataStreams() YDataLogger
datalogger→dataStreams()
`datalogger.get_dataStreams()`

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

int **get_dataStreams**(ArrayList<YDataStream> **v**)

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

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

Parameters :

v an array of YDataStream objects to be filled in

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

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

String **get_errorMessage**( )

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

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

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

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

```
int 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() YDataLogger
datalogger→friendlyName()
datalogger.get_friendlyName()

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

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

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

String **get_functionDescriptor**( )

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

Returns :

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

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

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

String **get_functionId**( )

For example `relay1`

Returns :

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

datalogger→get_hardwareId() YDataLogger
datalogger→hardwareId()
datalogger.get_hardwareId()

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

String **get_hardwareId**( )

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

Returns :

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

datalogger→get_logicalName() YDataLogger
datalogger→logicalName()
datalogger.get_logicalName()

Returns the logical name of the data logger.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the data logger. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

datalogger→get_module() YDataLogger
datalogger→module()`datalogger.get_module()`

Gets the `YModule` object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :
an instance of `YModule`

datalogger→get_recording() YDataLogger
datalogger→recording()
datalogger.get_recording()

Returns the current activation state of the data logger.

int **get_recording**( )

Returns :

either Y_RECORDING_OFF or Y_RECORDING_ON, according to the current activation state of the data logger

On failure, throws an exception or returns Y_RECORDING_INVALID.

datalogger→get_timeUTC() YDataLogger
datalogger→timeUTC()`datalogger.get_timeUTC()`

Returns the Unix timestamp for current UTC time, if known.

long **get_timeUTC**( )

Returns :
an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns `Y_TIMEUTC_INVALID`.

datalogger→get_userData() YDataLogger
datalogger→userData()
datalogger.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

Returns :
`true` if the data logger can be reached, and `false` otherwise

datalogger→load()`datalogger.load()` YDataLogger

Preloads the data logger cache with a specified validity duration.

int **load**(long **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→nextDataLogger() YDataLogger
`datalogger.nextDataLogger()`

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

YDataLogger **nextDataLogger**( )

Returns :

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

datalogger→registerValueCallback() YDataLogger
datalogger.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

datalogger→set_autoStart() YDataLogger
datalogger→setAutoStart()
datalogger.set_autoStart()

Changes the default activation state of the data logger on power up.

int **set_autoStart**(int **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() YDataLogger
datalogger→setLogicalName()
datalogger.set_logicalName()

Changes the logical name of the data logger.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the data logger.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

datalogger→set_recording() YDataLogger
datalogger→setRecording()
datalogger.set_recording()

Changes the activation state of the data logger to start/stop recording data.

int **set_recording**(int **newval**)

Parameters :

newval either Y_RECORDING_OFF or Y_RECORDING_ON, according to the activation state of the data logger to start/stop recording data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_timeUTC() YDataLogger
datalogger→setTimeUTC()
datalogger.set_timeUTC()

Changes the current UTC time reference used for recorded data.

int **set_timeUTC**(long **newval**)

Parameters :

newval an integer corresponding to the current UTC time reference used for recorded data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_userData() YDataLogger
datalogger→setUserData()
datalogger.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.9. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_datalogger.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDataLogger = yoctolib.YDataLogger;
php     require_once('yocto_datalogger.php');
cpp     #include "yocto_datalogger.h"
m       #import "yocto_datalogger.h"
pas     uses yocto_datalogger;
vb      yocto_datalogger.vb
cs      yocto_datalogger.cs
java    import com.yoctopuce.YoctoAPI.YDataLogger;
py      from yocto_datalogger import *
```

YDataRun methods
datarun→get_averageValue(**measureName**, **pos**) Returns the average value of the measure observed at the specified time period.
datarun→get_duration() Returns the duration (in seconds) of the data run.
datarun→get_maxValue(**measureName**, **pos**) Returns the maximal value of the measure observed at the specified time period.
datarun→get_measureNames() Returns the names of the measures recorded by the data logger.
datarun→get_minValue(**measureName**, **pos**) Returns the minimal value of the measure observed at the specified time period.
datarun→get_startTimeUTC() Returns the start time of the data run, relative to the Jan 1, 1970.
datarun→get_valueCount() Returns the number of values accessible in this run, given the selected data samples interval.
datarun→get_valueInterval() Returns the number of seconds covered by each value in this run.
datarun→set_valueInterval(**valueInterval**) Changes the number of seconds covered by each value in this run.

datarun→get_averageValue() YDataRun
datarun→averageValue()
datarun.get_averageValue()

Returns the average value of the measure observed at the specified time period.

double **get_averageValue**(String **measureName**, int **pos**)

datarun→get_averageValue()
datarun→averageValue()datarun.get_averageValue()

Returns the average value of the measure observed at the specified time period.

js function **get_averageValue**(**measureName**, **pos**)
nodejs function **get_averageValue**(**measureName**, **pos**)
php function **get_averageValue**(**$measureName**, **$pos**)
java double **get_averageValue**(String **measureName**, int **pos**)
py def **get_averageValue**(**measureName**, **pos**)

Parameters :

measureName	the name of the desired measure (one of the names returned by get_measureNames)
pos	the position index, between 0 and the value returned by get_valueCount

Returns :

a floating point number (the average value)

On failure, throws an exception or returns Y_AVERAGEVALUE_INVALID.

datarun→get_duration() YDataRun
datarun→duration()`datarun.get_duration()`

Returns the duration (in seconds) of the data run.

long **get_duration**( )

datarun→get_duration()
datarun→duration()`datarun.get_duration()`

Returns the duration (in seconds) of the data run.

`js` function **get_duration**( )
`nodejs` function **get_duration**( )
`php` function **get_duration**( )
`java` long **get_duration**( )
`py` def **get_duration**( )

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

Returns :
an unsigned number corresponding to the number of seconds between the beginning of the run (when the module was powered up) and the last recorded measure.

datarun→get_maxValue() YDataRun
datarun→maxValue()datarun.get_maxValue()

Returns the maximal value of the measure observed at the specified time period.

double **get_maxValue**(String **measureName**, int **pos**)

datarun→get_maxValue()
datarun→maxValue()datarun.get_maxValue()

Returns the maximal value of the measure observed at the specified time period.

```
js      function get_maxValue( measureName, pos)
nodejs  function get_maxValue( measureName, pos)
php     function get_maxValue( $measureName, $pos)
java    double get_maxValue( String measureName, int pos)
py      def get_maxValue( measureName, pos)
```

Parameters :

measureName	the name of the desired measure (one of the names returned by `get_measureNames`)
pos	the position index, between 0 and the value returned by `get_valueCount`

Returns :

a floating point number (the maximal value)

On failure, throws an exception or returns Y_MAXVALUE_INVALID.

datarun→get_measureNames() YDataRun
datarun→measureNames()
datarun.get_measureNames()

Returns the names of the measures recorded by the data logger.

ArrayList<String> **get_measureNames**( )

datarun→get_measureNames()
datarun→measureNames()datarun.get_measureNames()

Returns the names of the measures recorded by the data logger.

js	function **get_measureNames**( )
nodejs	function **get_measureNames**( )
php	function **get_measureNames**( )
java	ArrayList<String> **get_measureNames**( )
py	def **get_measureNames**( )

In most case, the measure names match the hardware identifier of the sensor that produced the data.

Returns :

a list of strings (the measure names) On failure, throws an exception or returns an empty array.

datarun→get_minValue() YDataRun
datarun→minValue()datarun.get_minValue()

Returns the minimal value of the measure observed at the specified time period.

double **get_minValue**(String **measureName**, int **pos**)

datarun→get_minValue()
datarun→minValue()datarun.get_minValue()

Returns the minimal value of the measure observed at the specified time period.

```
js      function get_minValue( measureName, pos)
nodejs  function get_minValue( measureName, pos)
php     function get_minValue( $measureName, $pos)
java    double get_minValue( String measureName, int pos)
py      def get_minValue( measureName, pos)
```

Parameters :

measureName	the name of the desired measure (one of the names returned by `get_measureNames`)
pos	the position index, between 0 and the value returned by `get_valueCount`

Returns :

a floating point number (the minimal value)

On failure, throws an exception or returns Y_MINVALUE_INVALID.

datarun→get_startTimeUTC() YDataRun
datarun→startTimeUTC()

Returns the start time of the data run, relative to the Jan 1, 1970.

If the UTC time was not set in the datalogger at any time during the recording of this data run, and if this is not the current run, this method returns 0.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data run (i.e. Unix time representation of the absolute time).

datarun→get_valueCount() YDataRun
datarun→valueCount()`datarun.get_valueCount()`

Returns the number of values accessible in this run, given the selected data samples interval.

int **get_valueCount**( )

datarun→get_valueCount()
datarun→valueCount()`datarun.get_valueCount()`

Returns the number of values accessible in this run, given the selected data samples interval.

`js` function **get_valueCount**( )
`nodejs` function **get_valueCount**( )
`php` function **get_valueCount**( )
`java` int **get_valueCount**( )
`py` def **get_valueCount**( )

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

Returns :
an unsigned number corresponding to the run duration divided by the samples interval.

datarun→get_valueInterval() YDataRun
datarun→valueInterval()
`datarun.get_valueInterval()`

Returns the number of seconds covered by each value in this run.

int **get_valueInterval**( )

datarun→get_valueInterval()
datarun→valueInterval()`datarun.get_valueInterval()`

Returns the number of seconds covered by each value in this run.

`js` function **get_valueInterval**( )
`nodejs` function **get_valueInterval**( )
`php` function **get_valueInterval**( )
`java` int **get_valueInterval**( )
`py` def **get_valueInterval**( )

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Returns :
an unsigned number corresponding to a number of seconds covered by each data sample in the Run.

datarun→set_valueInterval() YDataRun
datarun→setValueInterval()
datarun.set_valueInterval()

Changes the number of seconds covered by each value in this run.

void **set_valueInterval**(int **valueInterval**)

datarun→set_valueInterval()
datarun→setValueInterval()datarun.set_valueInterval()

Changes the number of seconds covered by each value in this run.

js	function **set_valueInterval**(**valueInterval**)
nodejs	function **set_valueInterval**(**valueInterval**)
php	function **set_valueInterval**(**$valueInterval**)
java	void **set_valueInterval**(int **valueInterval**)
py	def **set_valueInterval**(**valueInterval**)

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Parameters :

valueInterval an integer number of seconds.

Returns :

nothing

3.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instanciated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
```

YDataSet methods
dataset→get_endTimeUTC() Returns the end time of the dataset, relative to the Jan 1, 1970.
dataset→get_functionId() Returns the hardware identifier of the function that performed the measure, without reference to the module.
dataset→get_hardwareId() Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.
dataset→get_measures() Returns all measured values currently available for this DataSet, as a list of YMeasure objects.
dataset→get_preview() Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.
dataset→get_progress() Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.
dataset→get_startTimeUTC() Returns the start time of the dataset, relative to the Jan 1, 1970.
dataset→get_summary() Returns an YMeasure object which summarizes the whole DataSet.
dataset→get_unit() Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(**callback**, **context**)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→get_endTimeUTC() YDataSet
dataset→endTimeUTC()
dataset.get_endTimeUTC()

Returns the end time of the dataset, relative to the Jan 1, 1970.

long **get_endTimeUTC**( )

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

dataset→get_functionId() YDataSet
dataset→functionId()`dataset.get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

String **get_functionId**( )

For example `temperature1`.

Returns :

a string that identifies the function (ex: `temperature1`)

dataset→get_hardwareId() YDataSet
dataset→hardwareId()dataset.get_hardwareId()

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example THRMCPL1-123456.temperature1)

Returns :

a string that uniquely identifies the function (ex: THRMCPL1-123456.temperature1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

dataset→get_measures() YDataSet
dataset→measures()`dataset.get_measures()`

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

ArrayList<YMeasure> **get_measures**( )

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call loadMore() several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :
a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→get_preview() YDataSet
dataset→preview()dataset.get_preview()

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

ArrayList<YMeasure> **get_preview**( )

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :
a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_progress() YDataSet
dataset→progress()dataset.get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

```
int 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() YDataSet
dataset→startTimeUTC()
dataset.get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

long **get_startTimeUTC**( )

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

dataset→get_summary() YDataSet
dataset→summary()dataset.get_summary()

Returns an YMeasure object which summarizes the whole DataSet.

YMeasure **get_summary**( )

In includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

Returns :
an YMeasure object

dataset→get_unit() YDataSet
dataset→unit()`dataset.get_unit()`

Returns the measuring unit for the measured value.

String **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.

int **loadMore**( )

Returns :
an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

3.11. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
```

YDataStream methods
datastream→get_averageValue() Returns the average of all measures observed within this stream.
datastream→get_columnCount() Returns the number of data columns present in this stream.
datastream→get_columnNames() Returns the title (or meaning) of each data column present in this stream.
datastream→get_data(**row**, **col**) Returns a single measure from the data stream, specified by its row and column index.
datastream→get_dataRows() Returns the whole data set contained in the stream, as a bidimensional table of numbers.
datastream→get_dataSamplesIntervalMs() Returns the number of milliseconds between two consecutive rows of this data stream.
datastream→get_duration() Returns the approximate duration of this stream, in seconds.
datastream→get_maxValue() Returns the largest measure observed within this stream.
datastream→get_minValue() Returns the smallest measure observed within this stream.
datastream→get_rowCount() Returns the number of data rows present in this stream.
datastream→get_runIndex() Returns the run index of the data stream.
datastream→get_startTime() Returns the relative start time of the data stream, measured in seconds.
datastream→get_startTimeUTC()

Returns the start time of the data stream, relative to the Jan 1, 1970.

datastream→get_averageValue() YDataStream
datastream→averageValue()
datastream.get_averageValue()

Returns the average of all measures observed within this stream.

double **get_averageValue**( )

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_columnCount() YDataStream
datastream→columnCount()
datastream.get_columnCount()

Returns the number of data columns present in this stream.

int **get_columnCount**( )

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

datastream→get_columnNames() YDataStream
datastream→columnNames()
`datastream.get_columnNames()`

Returns the title (or meaning) of each data column present in this stream.

ArrayList<String> **get_columnNames**( )

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes _min, _avg and _max respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
a list containing as many strings as there are columns in the data stream.

On failure, throws an exception or returns an empty array.

datastream→get_data() YDataStream
datastream→data()`datastream.get_data()`

Returns a single measure from the data stream, specified by its row and column index.

double **get_data**(int **row**, int **col**)

The meaning of the values present in each column can be obtained using the method get_columnNames().

This method fetches the whole data stream from the device, if not yet done.

Parameters :

row row index

col column index

Returns :

a floating-point number

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_dataRows() YDataStream
datastream→dataRows()
datastream.get_dataRows()

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

```
ArrayList<ArrayList<Double>> get_dataRows( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Returns :

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.

datastream→get_dataSamplesIntervalMs() YDataStream
datastream→dataSamplesIntervalMs()
`datastream.get_dataSamplesIntervalMs()`

Returns the number of milliseconds between two consecutive rows of this data stream.

int **get_dataSamplesIntervalMs**( )

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

Returns :
an unsigned number corresponding to a number of milliseconds.

datastream→get_duration() YDataStream
datastream→duration()
datastream.get_duration()

Returns the approximate duration of this stream, in seconds.

int **get_duration**( )

Returns :
the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

datastream→get_maxValue() YDataStream
datastream→maxValue()
datastream.get_maxValue()

Returns the largest measure observed within this stream.

double **get_maxValue**( )

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :
a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_minValue() YDataStream
datastream→minValue()
datastream.get_minValue()

Returns the smallest measure observed within this stream.

double **get_minValue**( )

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the smallest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_rowCount() YDataStream
datastream→rowCount()
`datastream.get_rowCount()`

Returns the number of data rows present in this stream.

int **get_rowCount**( )

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

datastream→get_runIndex() YDataStream
datastream→runIndex()
datastream.get_runIndex()

Returns the run index of the data stream.

```
int get_runIndex( )
```

A run can be made of multiple datastreams, for different time intervals.

Returns :
an unsigned number corresponding to the run index.

datastream→get_startTime() YDataStream
datastream→startTime()
datastream.get_startTime()

Returns the relative start time of the data stream, measured in seconds.

int **get_startTime**( )

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `get_startTimeUTC()`.

Returns :

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

datastream→get_startTimeUTC() YDataStream
datastream→startTimeUTC()
`datastream.get_startTimeUTC()`

Returns the start time of the data stream, relative to the Jan 1, 1970.

long **get_startTimeUTC**( )

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data stream (i.e. Unix time representation of the absolute time).

3.12. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_digitalio.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDigitalIO = yoctolib.YDigitalIO;
php     require_once('yocto_digitalio.php');
cpp     #include "yocto_digitalio.h"
m       #import "yocto_digitalio.h"
pas     uses yocto_digitalio;
vb      yocto_digitalio.vb
cs      yocto_digitalio.cs
java    import com.yoctopuce.YoctoAPI.YDigitalIO;
py      from yocto_digitalio import *
```

Global functions
yFindDigitalIO(**func**) Retrieves a digital IO port for a given identifier.
yFirstDigitalIO() Starts the enumeration of digital IO ports currently accessible.
`YDigitalIO` methods
digitalio→delayedPulse(**bitno**, **ms_delay**, **ms_duration**) Schedules a pulse on a single bit for a specified duration.
digitalio→describe() Returns a short text that describes unambiguously the instance of the digital IO port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
digitalio→get_advertisedValue() Returns the current value of the digital IO port (no more than 6 characters).
digitalio→get_bitDirection(**bitno**) Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).
digitalio→get_bitOpenDrain(**bitno**) Returns the type of electrical interface of a single bit from the I/O port.
digitalio→get_bitPolarity(**bitno**) Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).
digitalio→get_bitState(**bitno**) Returns the state of a single bit of the I/O port.
digitalio→get_errorMessage() Returns the error message of the latest error with the digital IO port.
digitalio→get_errorType() Returns the numerical error code of the latest error with the digital IO port.
digitalio→get_friendlyName() Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

digitalio→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

digitalio→get_functionId()

Returns the hardware identifier of the digital IO port, without reference to the module.

digitalio→get_hardwareId()

Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID.

digitalio→get_logicalName()

Returns the logical name of the digital IO port.

digitalio→get_module()

Gets the YModule object for the device on which the function is located.

digitalio→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

digitalio→get_outputVoltage()

Returns the voltage source used to drive output bits.

digitalio→get_portDirection()

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→get_portOpenDrain()

Returns the electrical interface for each bit of the port.

digitalio→get_portPolarity()

Returns the polarity of all the bits of the port.

digitalio→get_portSize()

Returns the number of bits implemented in the I/O port.

digitalio→get_portState()

Returns the digital IO port state: bit 0 represents input 0, and so on.

digitalio→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

digitalio→isOnline()

Checks if the digital IO port is currently reachable, without raising any error.

digitalio→isOnline_async(**callback**, **context**)

Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

digitalio→load(**msValidity**)

Preloads the digital IO port cache with a specified validity duration.

digitalio→load_async(**msValidity**, **callback**, **context**)

Preloads the digital IO port cache with a specified validity duration (asynchronous version).

digitalio→nextDigitalIO()

Continues the enumeration of digital IO ports started using yFirstDigitalIO().

digitalio→pulse(**bitno**, **ms_duration**)

Triggers a pulse on a single bit for a specified duration.

digitalio→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

digitalio→set_bitDirection(**bitno**, **bitdirection**)

Changes the direction of a single bit from the I/O port.

digitalio→set_bitOpenDrain(**bitno**, **opendrain**)

Changes the electrical interface of a single bit from the I/O port.

digitalio→set_bitPolarity(**bitno**, **bitpolarity**)

Changes the polarity of a single bit from the I/O port.

digitalio→set_bitState(bitno, bitstate)

Sets a single bit of the I/O port.

digitalio→set_logicalName(newval)

Changes the logical name of the digital IO port.

digitalio→set_outputVoltage(newval)

Changes the voltage source used to drive output bits.

digitalio→set_portDirection(newval)

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→set_portOpenDrain(newval)

Changes the electrical interface for each bit of the port.

digitalio→set_portPolarity(newval)

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→set_portState(newval)

Changes the digital IO port state: bit 0 represents input 0, and so on.

digitalio→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

digitalio→toggle_bitState(bitno)

Reverts a single bit of the I/O port.

digitalio→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDigitalIO.FindDigitalIO() YDigitalIO yFindDigitalIO()YDigitalIO.FindDigitalIO()

Retrieves a digital IO port for a given identifier.

YDigitalIO **FindDigitalIO**(String **func**)

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() YDigitalIO yFirstDigitalIO()YDigitalIO.FirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

YDigitalIO **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() YDigitalIO
digitalio.delayedPulse()

Schedules a pulse on a single bit for a specified duration.

int **delayedPulse**(int **bitno**, int **ms_delay**, int **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`.

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 digital IO port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

digitalio→get_advertisedValue() YDigitalIO
digitalio→advertisedValue()
digitalio.get_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the digital IO port (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

digitalio→get_bitDirection() YDigitalIO
digitalio→bitDirection()
digitalio.get_bitDirection()

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

int **get_bitDirection**(int **bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→get_bitOpenDrain() YDigitalIO
digitalio→bitOpenDrain()
digitalio.get_bitOpenDrain()

Returns the type of electrical interface of a single bit from the I/O port.

int **get_bitOpenDrain**(int **bitno**)

(0 means the bit is an input, 1 an output).

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

0 means the a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

On failure, throws an exception or returns a negative error code.

digitalio→get_bitPolarity() YDigitalIO
digitalio→bitPolarity()
digitalio.get_bitPolarity()

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

int **get_bitPolarity**(int **bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→get_bitState() YDigitalIO
digitalio→bitState()`digitalio.get_bitState()`

Returns the state of a single bit of the I/O port.

int **get_bitState**(int **bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

the bit state (0 or 1)

On failure, throws an exception or returns a negative error code.

digitalio→get_errorMessage() YDigitalIO
digitalio→errorMessage()
digitalio.get_errorMessage()

Returns the error message of the latest error with the digital IO port.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the digital IO port object

digitalio→get_errorType() YDigitalIO
digitalio→errorType()`digitalio.get_errorType()`

Returns the numerical error code of the latest error with the digital IO port.

```
int 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() YDigitalIO
digitalio→friendlyName()
digitalio.get_friendlyName()

Returns a global identifier of the digital IO port in the format MODULE_NAME.FUNCTION_NAME.

String **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() YDigitalIO
digitalio→functionDescriptor()
digitalio.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

digitalio→get_functionId() YDigitalIO
digitalio→functionId()
digitalio.get_functionId()

Returns the hardware identifier of the digital IO port, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the digital IO port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

digitalio→get_hardwareId() YDigitalIO
digitalio→hardwareId()
digitalio.get_hardwareId()

Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the digital IO port. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the digital IO port (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

digitalio→get_logicalName() YDigitalIO
digitalio→logicalName()
digitalio.get_logicalName()

Returns the logical name of the digital IO port.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the digital IO port. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

digitalio→get_module() YDigitalIO
digitalio→module()`digitalio.get_module()`

Gets the `YModule` object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :
an instance of `YModule`

digitalio→get_outputVoltage() YDigitalIO
digitalio→outputVoltage()
digitalio.get_outputVoltage()

Returns the voltage source used to drive output bits.

int **get_outputVoltage**( )

Returns :

a value among Y_OUTPUTVOLTAGE_USB_5V, Y_OUTPUTVOLTAGE_USB_3V and Y_OUTPUTVOLTAGE_EXT_V corresponding to the voltage source used to drive output bits

On failure, throws an exception or returns Y_OUTPUTVOLTAGE_INVALID.

digitalio→get_portDirection() YDigitalIO
digitalio→portDirection()
digitalio.get_portDirection()

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

int **get_portDirection**( )

Returns :

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

On failure, throws an exception or returns `Y_PORTDIRECTION_INVALID`.

digitalio→get_portOpenDrain() YDigitalIO
digitalio→portOpenDrain()
digitalio.get_portOpenDrain()

Returns the electrical interface for each bit of the port.

int **get_portOpenDrain**( )

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

Returns :
an integer corresponding to the electrical interface for each bit of the port

On failure, throws an exception or returns Y_PORTOPENDRAIN_INVALID.

digitalio→get_portPolarity() YDigitalIO
digitalio→portPolarity()
digitalio.get_portPolarity()

Returns the polarity of all the bits of the port.

int **get_portPolarity**( )

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

Returns :
an integer corresponding to the polarity of all the bits of the port

On failure, throws an exception or returns Y_PORTPOLARITY_INVALID.

digitalio→get_portSize() YDigitalIO
digitalio→portSize()`digitalio.get_portSize()`

Returns the number of bits implemented in the I/O port.

int **get_portSize**( )

Returns :
an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns `Y_PORTSIZE_INVALID`.

digitalio→get_portState() YDigitalIO
digitalio→portState()`digitalio.get_portState()`

Returns the digital IO port state: bit 0 represents input 0, and so on.

int **get_portState**( )

Returns :
an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

On failure, throws an exception or returns `Y_PORTSTATE_INVALID`.

digitalio→get_userData() YDigitalIO
digitalio→userData()`digitalio.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the digital IO port.

Returns :

`true` if the digital IO port can be reached, and `false` otherwise

digitalio→load()`digitalio.load()` YDigitalIO

Preloads the digital IO port cache with a specified validity duration.

int **load**(long **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→nextDigitalIO() YDigitalIO
digitalio.nextDigitalIO()

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

YDigitalIO **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.

int **pulse**(int **bitno**, int **ms_duration**)

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno	the bit number; lowest bit has index 0
ms_duration	desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→registerValueCallback() YDigitalIO
digitalio.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

digitalio→set_bitDirection() YDigitalIO
digitalio→setBitDirection()
digitalio.set_bitDirection()

Changes the direction of a single bit from the I/O port.

int **set_bitDirection**(int **bitno**, int **bitdirection**)

Parameters :

bitno the bit number; lowest bit has index 0

bitdirection direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_bitOpenDrain() YDigitalIO
digitalio→setBitOpenDrain()
digitalio.set_bitOpenDrain()

Changes the electrical interface of a single bit from the I/O port.

int **set_bitOpenDrain**(int **bitno**, int **opendrain**)

Parameters :

bitno the bit number; lowest bit has index 0

opendrain 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_bitPolarity() YDigitalIO
digitalio→setBitPolarity()
digitalio.set_bitPolarity()

Changes the polarity of a single bit from the I/O port.

int **set_bitPolarity**(int **bitno**, int **bitpolarity**)

Parameters :

bitno the bit number; lowest bit has index 0.

bitpolarity polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_bitState() YDigitalIO
digitalio→setBitState()`digitalio.set_bitState()`

Sets a single bit of the I/O port.

int **set_bitState**(int **bitno**, int **bitstate**)

Parameters :

bitno the bit number; lowest bit has index 0

bitstate the state of the bit (1 or 0)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_logicalName() YDigitalIO
digitalio→setLogicalName()
digitalio.set_logicalName()

Changes the logical name of the digital IO port.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the digital IO port.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

digitalio→set_outputVoltage() YDigitalIO
digitalio→setOutputVoltage()
digitalio.set_outputVoltage()

Changes the voltage source used to drive output bits.

int **set_outputVoltage**(int **newval**)

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portDirection() YDigitalIO
digitalio→setPortDirection()
digitalio.set_portDirection()

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

int **set_portDirection**(int **newval**)

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portOpenDrain() YDigitalIO
digitalio→setPortOpenDrain()
digitalio.set_portOpenDrain()

Changes the electrical interface for each bit of the port.

int **set_portOpenDrain**(int **newval**)

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the electrical interface for each bit of the port

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portPolarity() YDigitalIO
digitalio→setPortPolarity()
digitalio.set_portPolarity()

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

int **set_portPolarity**(int **newval**)

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

Parameters :

newval an integer corresponding to the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portState() YDigitalIO
digitalio→setPortState()
digitalio.set_portState()

Changes the digital IO port state: bit 0 represents input 0, and so on.

int **set_portState**(int **newval**)

This function has no effect on bits configured as input in `portDirection`.

Parameters :

newval an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_userData() YDigitalIO
digitalio→setUserData()
digitalio.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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() YDigitalIO
digitalio.toggle_bitState()

Reverts a single bit of the I/O port.

int **toggle_bitState**(int **bitno**)

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.13. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_display.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDisplay = yoctolib.YDisplay;
php     require_once('yocto_display.php');
cpp     #include "yocto_display.h"
m       #import "yocto_display.h"
pas     uses yocto_display;
vb      yocto_display.vb
cs      yocto_display.cs
java    import com.yoctopuce.YoctoAPI.YDisplay;
py      from yocto_display import *
```

Global functions
yFindDisplay(**func**) Retrieves a display for a given identifier.
yFirstDisplay() Starts the enumeration of displays currently accessible.
`YDisplay` methods
display→copyLayerContent(**srcLayerId**, **dstLayerId**) Copies the whole content of a layer to another layer.
display→describe() Returns a short text that describes unambiguously the instance of the display in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
display→fade(**brightness**, **duration**) Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.
display→get_advertisedValue() Returns the current value of the display (no more than 6 characters).
display→get_brightness() Returns the luminosity of the module informative leds (from 0 to 100).
display→get_displayHeight() Returns the display height, in pixels.
display→get_displayLayer(**layerId**) Returns a YDisplayLayer object that can be used to draw on the specified layer.
display→get_displayType() Returns the display type: monochrome, gray levels or full color.
display→get_displayWidth() Returns the display width, in pixels.
display→get_enabled() Returns true if the screen is powered, false otherwise.
display→get_errorMessage() Returns the error message of the latest error with the display.

display→get_errorType()
Returns the numerical error code of the latest error with the display.

display→get_friendlyName()
Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

display→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

display→get_functionId()
Returns the hardware identifier of the display, without reference to the module.

display→get_hardwareId()
Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

display→get_layerCount()
Returns the number of available layers to draw on.

display→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

display→get_logicalName()
Returns the logical name of the display.

display→get_module()
Gets the `YModule` object for the device on which the function is located.

display→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

display→get_orientation()
Returns the currently selected display orientation.

display→get_startupSeq()
Returns the name of the sequence to play when the displayed is powered on.

display→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

display→isOnline()
Checks if the display is currently reachable, without raising any error.

display→isOnline_async(callback, context)
Checks if the display is currently reachable, without raising any error (asynchronous version).

display→load(msValidity)
Preloads the display cache with a specified validity duration.

display→load_async(msValidity, callback, context)
Preloads the display cache with a specified validity duration (asynchronous version).

display→newSequence()
Starts to record all display commands into a sequence, for later replay.

display→nextDisplay()
Continues the enumeration of displays started using `yFirstDisplay()`.

display→pauseSequence(delay_ms)
Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

display→playSequence(sequenceName)
Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

display→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

display→resetAll()

Clears the display screen and resets all display layers to their default state.

display→saveSequence(**sequenceName**)

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

display→set_brightness(**newval**)

Changes the brightness of the display.

display→set_enabled(**newval**)

Changes the power state of the display.

display→set_logicalName(**newval**)

Changes the logical name of the display.

display→set_orientation(**newval**)

Changes the display orientation.

display→set_startupSeq(**newval**)

Changes the name of the sequence to play when the displayed is powered on.

display→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

display→stopSequence()

Stops immediately any ongoing sequence replay.

display→swapLayerContent(**layerIdA**, **layerIdB**)

Swaps the whole content of two layers.

display→upload(**pathname**, **content**)

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

display→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDisplay.FindDisplay() YDisplay yFindDisplay()YDisplay.FindDisplay()

Retrieves a display for a given identifier.

YDisplay **FindDisplay**(String **func**)

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() YDisplay yFirstDisplay()YDisplay.FirstDisplay()

Starts the enumeration of displays currently accessible.

YDisplay **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() YDisplay
display.copyLayerContent()

Copies the whole content of a layer to another layer.

int **copyLayerContent**(int **srcLayerId**, int **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.

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 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.

int **fade**(int **brightness**, int **duration**)

Parameters :

brightness the new screen brightness

duration duration of the brightness transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→get_advertisedValue() YDisplay
display→advertisedValue()
display.get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the display (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

display→get_brightness() YDisplay
display→brightness()`display.get_brightness()`

Returns the luminosity of the module informative leds (from 0 to 100).

int **get_brightness**( )

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_BRIGHTNESS_INVALID`.

display→get_displayHeight() YDisplay
display→displayHeight()
display.get_displayHeight()

Returns the display height, in pixels.

int **get_displayHeight**( )

Returns :
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

display→get_displayLayer() YDisplay
display→displayLayer()
display.get_displayLayer()

Returns a YDisplayLayer object that can be used to draw on the specified layer.

synchronized YDisplayLayer **get_displayLayer**(int **layerId**)

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

Parameters :

layerId the identifier of the layer (a number in range 0..layerCount-1)

Returns :

an `YDisplayLayer` object

On failure, throws an exception or returns `null`.

display→get_displayType() YDisplay
display→displayType()
display.get_displayType()

Returns the display type: monochrome, gray levels or full color.

int **get_displayType**( )

Returns :

a value among Y_DISPLAYTYPE_MONO, Y_DISPLAYTYPE_GRAY and Y_DISPLAYTYPE_RGB corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns Y_DISPLAYTYPE_INVALID.

display→get_displayWidth() YDisplay
display→displayWidth()
display.get_displayWidth()

Returns the display width, in pixels.

int **get_displayWidth**( )

Returns :
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

display→get_enabled() YDisplay
display→enabled()`display.get_enabled()`

Returns true if the screen is powered, false otherwise.

int **get_enabled**( )

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

display→get_errorMessage() YDisplay
display→errorMessage()
display.get_errorMessage()

Returns the error message of the latest error with the display.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the display object

display→get_errorType() YDisplay
display→errorType()`display.get_errorType()`

Returns the numerical error code of the latest error with the display.

int **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() YDisplay
display→friendlyName()
display.get_friendlyName()

Returns a global identifier of the display in the format MODULE_NAME.FUNCTION_NAME.

String **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() YDisplay
display→functionDescriptor()
display.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

display→get_functionId() YDisplay
display→functionId()`display.get_functionId()`

Returns the hardware identifier of the display, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the display (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

display→get_hardwareId() YDisplay
display→hardwareId()display.get_hardwareId()

Returns the unique hardware identifier of the display in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the display (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

display→get_layerCount() YDisplay
display→layerCount()`display.get_layerCount()`

Returns the number of available layers to draw on.

int **get_layerCount**( )

Returns :
an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns `Y_LAYERCOUNT_INVALID`.

display→get_layerHeight() YDisplay
display→layerHeight()
display.get_layerHeight()

Returns the height of the layers to draw on, in pixels.

int **get_layerHeight**( )

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

display→get_layerWidth() YDisplay
display→layerWidth()`display.get_layerWidth()`

Returns the width of the layers to draw on, in pixels.

int **get_layerWidth**( )

Returns :
an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns `Y_LAYERWIDTH_INVALID`.

display→get_logicalName() YDisplay
display→logicalName()
display.get_logicalName()

Returns the logical name of the display.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the display. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

display→get_module() YDisplay
display→module()`display.get_module()`

Gets the `YModule` object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :
an instance of `YModule`

display→get_orientation() YDisplay
display→orientation()`display.get_orientation()`

Returns the currently selected display orientation.

int **get_orientation**( )

Returns :

a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the currently selected display orientation

On failure, throws an exception or returns `Y_ORIENTATION_INVALID`.

display→get_startupSeq() YDisplay
display→startupSeq()`display.get_startupSeq()`

Returns the name of the sequence to play when the displayed is powered on.

String **get_startupSeq**( )

Returns :
a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns `Y_STARTUPSEQ_INVALID`.

display→get_userData() YDisplay
display→userData()`display.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

Returns :
`true` if the display can be reached, and `false` otherwise

display→load()`display.load()` YDisplay

Preloads the display cache with a specified validity duration.

int **load**(long **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→newSequence()`display.newSequence()` YDisplay

Starts to record all display commands into a sequence, for later replay.

```
int 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()`.

YDisplay **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() YDisplay
display.pauseSequence()

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

int **pauseSequence**(int **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()`.

int **playSequence**(String **sequenceName**)

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→registerValueCallback() YDisplay
display.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback 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.

int **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.

int **saveSequence**(String **sequenceName**)

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_brightness() YDisplay
display→setBrightness()
display.set_brightness()

Changes the brightness of the display.

```
int set_brightness( int newval)
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the brightness of the display

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_enabled() YDisplay
display→setEnabled()`display.set_enabled()`

Changes the power state of the display.

int **set_enabled**(int **newval**)

Parameters :

newval either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the power state of the display

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_logicalName() YDisplay
display→setLogicalName()
display.set_logicalName()

Changes the logical name of the display.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the display.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

display→set_orientation() YDisplay
display→setOrientation()
display.set_orientation()

Changes the display orientation.

int **set_orientation**(int **newval**)

Remember to call the saveToFlash() method of the module if the modification must be kept.

Parameters :

newval a value among Y_ORIENTATION_LEFT, Y_ORIENTATION_UP, Y_ORIENTATION_RIGHT and Y_ORIENTATION_DOWN corresponding to the display orientation

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_startupSeq() YDisplay
display→setStartupSeq()
display.set_startupSeq()

Changes the name of the sequence to play when the displayed is powered on.

int **set_startupSeq**(String **newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the name of the sequence to play when the displayed is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_userData() YDisplay
display→setUserData()`display.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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.

int **stopSequence**( )

The display is left as is.

Returns :
YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→swapLayerContent() YDisplay
display.swapLayerContent()

Swaps the whole content of two layers.

int **swapLayerContent**(int **layerIdA**, int **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.

int **upload**(String **pathname**)

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.14. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_display.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDisplay = yoctolib.YDisplay;
php     require_once('yocto_display.php');
cpp     #include "yocto_display.h"
m       #import "yocto_display.h"
pas     uses yocto_display;
vb      yocto_display.vb
cs      yocto_display.cs
java    import com.yoctopuce.YoctoAPI.YDisplay;
py      from yocto_display import *
```

YDisplayLayer methods
displaylayer→clear() Erases the whole content of the layer (makes it fully transparent).
displaylayer→clearConsole() Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.
displaylayer→consoleOut(**text**) Outputs a message in the console area, and advances the console pointer accordingly.
displaylayer→drawBar(**x1**, **y1**, **x2**, **y2**) Draws a filled rectangular bar at a specified position.
displaylayer→drawBitmap(**x**, **y**, **w**, **bitmap**, **bgcol**) Draws a bitmap at the specified position.
displaylayer→drawCircle(**x**, **y**, **r**) Draws an empty circle at a specified position.
displaylayer→drawDisc(**x**, **y**, **r**) Draws a filled disc at a given position.
displaylayer→drawImage(**x**, **y**, **imagename**) Draws a GIF image at the specified position.
displaylayer→drawPixel(**x**, **y**) Draws a single pixel at the specified position.
displaylayer→drawRect(**x1**, **y1**, **x2**, **y2**) Draws an empty rectangle at a specified position.
displaylayer→drawText(**x**, **y**, **anchor**, **text**) Draws a text string at the specified position.
displaylayer→get_display() Gets parent YDisplay.
displaylayer→get_displayHeight() Returns the display height, in pixels.
displaylayer→get_displayWidth() Returns the display width, in pixels.

displaylayer→get_layerHeight()

Returns the height of the layers to draw on, in pixels.

displaylayer→get_layerWidth()

Returns the width of the layers to draw on, in pixels.

displaylayer→hide()

Hides the layer.

displaylayer→lineTo(**x**, **y**)

Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(**x**, **y**)

Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()

Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(**color**)

Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

displaylayer→selectFont(**fontname**)

Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(**graylevel**)

Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(**mode**)

Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(**bgcol**)

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(**x1**, **y1**, **x2**, **y2**)

Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(**wordwrap**)

Sets up the wrapping behaviour used by the `consoleOut` function.

displaylayer→setLayerPosition(**x**, **y**, **scrollTime**)

Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()

Shows the layer.

displaylayer→clear()`displaylayer.clear()` YDisplayLayer

Erases the whole content of the layer (makes it fully transparent).

int **clear**( )

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset()` instead.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→clearConsole() YDisplayLayer
displaylayer.clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

int **clearConsole**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→consoleOut() YDisplayLayer
`displaylayer.consoleOut()`

Outputs a message in the console area, and advances the console pointer accordingly.

int **consoleOut**(String **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.

int **drawBar**(int **x1**, int **y1**, int **x2**, int **y2**)

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBitmap() YDisplayLayer
displaylayer.drawBitmap()

Draws a bitmap at the specified position.

int **drawBitmap**(int **x**, int **y**, int **w**, int **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() YDisplayLayer
`displaylayer.drawCircle()`

Draws an empty circle at a specified position.

int **drawCircle**(int **x**, int **y**, int **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() YDisplayLayer
`displaylayer.drawDisc()`

Draws a filled disc at a given position.

int **drawDisc**(int **x**, int **y**, int **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() YDisplayLayer
`displaylayer.drawImage()`

Draws a GIF image at the specified position.

int **drawImage**(int **x**, int **y**, String **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() YDisplayLayer
displaylayer.drawPixel()

Draws a single pixel at the specified position.

```
int drawPixel( int x, int 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() YDisplayLayer
displaylayer.drawRect()

Draws an empty rectangle at a specified position.

int **drawRect**(int **x1**, int **y1**, int **x2**, int **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() YDisplayLayer
displaylayer.drawText()

Draws a text string at the specified position.

int **drawText**(int **x**, int **y**, ALIGN **anchor**, String **text**)

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

Parameters :

x	the distance from left of layer to the text anchor point, in pixels
y	the distance from top of layer to the text anchor point, in pixels
anchor	the text anchor point, chosen among the Y_ALIGN enumeration: Y_ALIGN_TOP_LEFT, Y_ALIGN_CENTER_LEFT, Y_ALIGN_BASELINE_LEFT, Y_ALIGN_BOTTOM_LEFT, Y_ALIGN_TOP_CENTER, Y_ALIGN_CENTER, Y_ALIGN_BASELINE_CENTER, Y_ALIGN_BOTTOM_CENTER, Y_ALIGN_TOP_DECIMAL, Y_ALIGN_CENTER_DECIMAL, Y_ALIGN_BASELINE_DECIMAL, Y_ALIGN_BOTTOM_DECIMAL, Y_ALIGN_TOP_RIGHT, Y_ALIGN_CENTER_RIGHT, Y_ALIGN_BASELINE_RIGHT, Y_ALIGN_BOTTOM_RIGHT.
text	the text string to draw

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→get_display() YDisplayLayer
displaylayer→display()
displaylayer.get_display()

Gets parent YDisplay.

YDisplay **get_display**( )

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :
an `YDisplay` object

displaylayer→get_displayHeight() YDisplayLayer
displaylayer→displayHeight()
displaylayer.get_displayHeight()

Returns the display height, in pixels.

int **get_displayHeight**( )

Returns :

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth() YDisplayLayer
displaylayer→displayWidth()
displaylayer.get_displayWidth()

Returns the display width, in pixels.

```
int get_displayWidth( )
```

Returns :

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→get_layerHeight() YDisplayLayer
displaylayer→layerHeight()
displaylayer.get_layerHeight()

Returns the height of the layers to draw on, in pixels.

int **get_layerHeight**( )

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth() YDisplayLayer
displaylayer→layerWidth()
displaylayer.get_layerWidth()

Returns the width of the layers to draw on, in pixels.

int **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.

int **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.

int **lineTo**(int **x**, int **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.

int **moveTo**(int **x**, int **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).

int **reset**( )

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectColorPen() YDisplayLayer
displaylayer.selectColorPen()

Selects the pen color for all subsequent drawing functions, including text drawing.

int **selectColorPen**(int **color**)

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectEraser() YDisplayLayer
displaylayer.selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

int **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() YDisplayLayer
displaylayer.selectFont()

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```
int selectFont( String fontname)
```

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

Parameters :

fontname the font file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectGrayPen() YDisplayLayer
displaylayer.selectGrayPen()

Selects the pen gray level for all subsequent drawing functions, including text drawing.

int **selectGrayPen**(int **graylevel**)

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the lighest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

Parameters :

graylevel the desired gray level, from 0 to 255

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setAntialiasingMode() YDisplayLayer
displaylayer.setAntialiasingMode()

Enables or disables anti-aliasing for drawing oblique lines and circles.

int **setAntialiasingMode**(boolean **mode**)

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

Parameters :

mode true to enable antialiasing, false to disable it.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleBackground() YDisplayLayer
`displaylayer.setConsoleBackground()`

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

int **setConsoleBackground**(int **bgcol**)

Parameters :

bgcol the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleMargins() YDisplayLayer
displaylayer.setConsoleMargins()

Sets up display margins for the consoleOut function.

int **setConsoleMargins**(int **x1**, int **y1**, int **x2**, int **y2**)

Parameters :

x1 the distance from left of layer to the left margin, in pixels

y1 the distance from top of layer to the top margin, in pixels

x2 the distance from left of layer to the right margin, in pixels

y2 the distance from top of layer to the bottom margin, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleWordWrap() YDisplayLayer
`displaylayer.setConsoleWordWrap()`

Sets up the wrapping behaviour used by the `consoleOut` function.

int **setConsoleWordWrap**(boolean **wordwrap**)

Parameters :

wordwrap `true` to wrap only between words, `false` to wrap on the last column anyway.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setLayerPosition() YDisplayLayer
displaylayer.setLayerPosition()

Sets the position of the layer relative to the display upper left corner.

int **setLayerPosition**(int **x**, int **y**, int **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.

int **unhide**( )

Shows the layer again after a hide command.

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.15. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_dualpower.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDualPower = yoctolib.YDualPower;
php     require_once('yocto_dualpower.php');
cpp     #include "yocto_dualpower.h"
m       #import "yocto_dualpower.h"
pas     uses yocto_dualpower;
vb      yocto_dualpower.vb
cs      yocto_dualpower.cs
java    import com.yoctopuce.YoctoAPI.YDualPower;
py      from yocto_dualpower import *
```

Global functions
yFindDualPower(**func**) Retrieves a dual power control for a given identifier.
yFirstDualPower() Starts the enumeration of dual power controls currently accessible.
`YDualPower` methods
dualpower→describe() Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
dualpower→get_advertisedValue() Returns the current value of the power control (no more than 6 characters).
dualpower→get_errorMessage() Returns the error message of the latest error with the power control.
dualpower→get_errorType() Returns the numerical error code of the latest error with the power control.
dualpower→get_extVoltage() Returns the measured voltage on the external power source, in millivolts.
dualpower→get_friendlyName() Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.
dualpower→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
dualpower→get_functionId() Returns the hardware identifier of the power control, without reference to the module.
dualpower→get_hardwareId() Returns the unique hardware identifier of the power control in the form `SERIAL.FUNCTIONID`.
dualpower→get_logicalName() Returns the logical name of the power control.
dualpower→get_module()

Gets the YModule object for the device on which the function is located.

dualpower→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

dualpower→get_powerControl()

Returns the selected power source for module functions that require lots of current.

dualpower→get_powerState()

Returns the current power source for module functions that require lots of current.

dualpower→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

dualpower→isOnline()

Checks if the power control is currently reachable, without raising any error.

dualpower→isOnline_async(callback, context)

Checks if the power control is currently reachable, without raising any error (asynchronous version).

dualpower→load(msValidity)

Preloads the power control cache with a specified validity duration.

dualpower→load_async(msValidity, callback, context)

Preloads the power control cache with a specified validity duration (asynchronous version).

dualpower→nextDualPower()

Continues the enumeration of dual power controls started using yFirstDualPower().

dualpower→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

dualpower→set_logicalName(newval)

Changes the logical name of the power control.

dualpower→set_powerControl(newval)

Changes the selected power source for module functions that require lots of current.

dualpower→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

dualpower→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDualPower.FindDualPower() YDualPower yFindDualPower()YDualPower.FindDualPower()

Retrieves a dual power control for a given identifier.

YDualPower **FindDualPower**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDualPower.isOnline()` to test if the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

YDualPower.FirstDualPower() YDualPower yFirstDualPower()YDualPower.FirstDualPower()

Starts the enumeration of dual power controls currently accessible.

YDualPower **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.

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 power control (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

dualpower→get_advertisedValue() YDualPower
dualpower→advertisedValue()
dualpower.get_advertisedValue()

Returns the current value of the power control (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the power control (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

dualpower→get_errorMessage() YDualPower
dualpower→errorMessage()
dualpower.get_errorMessage()

Returns the error message of the latest error with the power control.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the power control object

dualpower→get_errorType() YDualPower
dualpower→errorType()
dualpower.get_errorType()

Returns the numerical error code of the latest error with the power control.

```
int 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() YDualPower
dualpower→extVoltage()
dualpower.get_extVoltage()

Returns the measured voltage on the external power source, in millivolts.

int **get_extVoltage**( )

Returns :
an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns Y_EXTVOLTAGE_INVALID.

dualpower→get_friendlyName() YDualPower
dualpower→friendlyName()
dualpower.get_friendlyName()

Returns a global identifier of the power control in the format MODULE_NAME.FUNCTION_NAME.

String **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() YDualPower
dualpower→functionDescriptor()
dualpower.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

dualpower→get_functionId() YDualPower
dualpower→functionId()
dualpower.get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the power control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

dualpower→get_hardwareId() YDualPower
dualpower→hardwareId()
dualpower.get_hardwareId()

Returns the unique hardware identifier of the power control in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the power control. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the power control (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

dualpower→get_logicalName() YDualPower
dualpower→logicalName()
dualpower.get_logicalName()

Returns the logical name of the power control.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the power control. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

dualpower→get_module() YDualPower
dualpower→module()dualpower.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

dualpower→get_powerControl() YDualPower
dualpower→powerControl()
dualpower.get_powerControl()

Returns the selected power source for module functions that require lots of current.

int **get_powerControl**( )

Returns :

a value among Y_POWERCONTROL_AUTO, Y_POWERCONTROL_FROM_USB, Y_POWERCONTROL_FROM_EXT and Y_POWERCONTROL_OFF corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERCONTROL_INVALID.

dualpower→get_powerState() YDualPower
dualpower→powerState()
dualpower.get_powerState()

Returns the current power source for module functions that require lots of current.

int **get_powerState**( )

Returns :

a value among Y_POWERSTATE_OFF, Y_POWERSTATE_FROM_USB and Y_POWERSTATE_FROM_EXT corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERSTATE_INVALID.

dualpower→get_userData() YDualPower
dualpower→userData()dualpower.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

```
boolean isOnline( )
```

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power control.

Returns :

`true` if the power control can be reached, and `false` otherwise

dualpower→load()dualpower.load() YDualPower

Preloads the power control cache with a specified validity duration.

int **load**(long **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→nextDualPower() YDualPower
dualpower.nextDualPower()

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

YDualPower **nextDualPower**( )

Returns :

a pointer to a `YDualPower` object, corresponding to a dual power control currently online, or a `null` pointer if there are no more dual power controls to enumerate.

dualpower→registerValueCallback() YDualPower
dualpower.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

dualpower→set_logicalName() YDualPower
dualpower→setLogicalName()
dualpower.set_logicalName()

Changes the logical name of the power control.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the power control.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

dualpower→set_powerControl() YDualPower
dualpower→setPowerControl()
dualpower.set_powerControl()

Changes the selected power source for module functions that require lots of current.

int **set_powerControl**(int **newval**)

Parameters :

newval a value among Y_POWERCONTROL_AUTO, Y_POWERCONTROL_FROM_USB, Y_POWERCONTROL_FROM_EXT and Y_POWERCONTROL_OFF corresponding to the selected power source for module functions that require lots of current

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

dualpower→set_userData() YDualPower
dualpower→setUserData()
dualpower.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.16. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_files.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YFiles = yoctolib.YFiles;
php     require_once('yocto_files.php');
cpp     #include "yocto_files.h"
m       #import "yocto_files.h"
pas     uses yocto_files;
vb      yocto_files.vb
cs      yocto_files.cs
java    import com.yoctopuce.YoctoAPI.YFiles;
py      from yocto_files import *
```

Global functions

yFindFiles(**func**)

Retrieves a filesystem for a given identifier.

yFirstFiles()

Starts the enumeration of filesystems currently accessible.

YFiles methods

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

files→download(**pathname**)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(**pathname**, **callback**, **context**)

Downloads the requested file and returns a binary buffer with its content.

files→format_fs()

Reinitializes the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

files→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

files→get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

files→get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

files→get_list(**pattern**)

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

files→get_logicalName()

Returns the logical name of the filesystem.

files→get_module()

Gets the `YModule` object for the device on which the function is located.

files→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

files→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

files→isOnline()

Checks if the filesystem is currently reachable, without raising any error.

files→isOnline_async(**callback**, **context**)

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

files→load(**msValidity**)

Preloads the filesystem cache with a specified validity duration.

files→load_async(**msValidity**, **callback**, **context**)

Preloads the filesystem cache with a specified validity duration (asynchronous version).

files→nextFiles()

Continues the enumeration of filesystems started using `yFirstFiles()`.

files→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

files→remove(**pathname**)

Deletes a file, given by its full path name, from the filesystem.

files→set_logicalName(**newval**)

Changes the logical name of the filesystem.

files→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

files→upload(**pathname**, **content**)

Uploads a file to the filesystem, to the specified full path name.

files→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YFiles.FindFiles() YFiles
yFindFiles()YFiles.FindFiles()

Retrieves a filesystem for a given identifier.

YFiles **FindFiles**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the filesystem

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles() YFiles
yFirstFiles()YFiles.FirstFiles()

Starts the enumeration of filesystems currently accessible.

YFiles **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`.

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 filesystem (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

files→format_fs()`files.format_fs()` YFiles

Reinitializes the filesystem to its clean, unfragmented, empty state.

int **format_fs**( )

All files previously uploaded are permanently lost.

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→get_advertisedValue() YFiles
files→advertisedValue()
files.get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the filesystem (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

files→get_errorMessage() YFiles
files→errorMessage()files.get_errorMessage()

Returns the error message of the latest error with the filesystem.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the filesystem object

files→get_errorType() YFiles
files→errorType()`files.get_errorType()`

Returns the numerical error code of the latest error with the filesystem.

int **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() YFiles
files→filesCount()files.get_filesCount()

Returns the number of files currently loaded in the filesystem.

int **get_filesCount**( )

Returns :
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns Y_FILESCOUNT_INVALID.

files→get_freeSpace() YFiles
files→freeSpace()`files.get_freeSpace()`

Returns the free space for uploading new files to the filesystem, in bytes.

int **get_freeSpace**( )

Returns :
an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns Y_FREESPACE_INVALID.

files→get_friendlyName() YFiles
files→friendlyName()files.get_friendlyName()

Returns a global identifier of the filesystem in the format MODULE_NAME.FUNCTION_NAME.

String **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() YFiles
files→functionDescriptor()
files.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

files→get_functionId() YFiles
files→functionId()files.get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the filesystem (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

files→get_hardwareId() YFiles
files→hardwareId()`files.get_hardwareId()`

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the filesystem (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

files→get_list() YFiles
files→list()files.get_list()

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

ArrayList<YFileRecord> **get_list**(String **pattern**)

Parameters :

pattern an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

Returns :

a list of `YFileRecord` objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName() YFiles
files→logicalName()`files.get_logicalName()`

Returns the logical name of the filesystem.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the filesystem. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

files→get_module() YFiles
files→module()files.get_module()

Gets the YModule object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

files→get_userData() YFiles
files→userData()`files.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

```
boolean isOnline( )
```

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

Returns :

`true` if the filesystem can be reached, and `false` otherwise

files→load()`files.load()` YFiles

Preloads the filesystem cache with a specified validity duration.

int **load**(long **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→nextFiles()`files.nextFiles()` YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

YFiles **nextFiles**( )

Returns :

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

files→registerValueCallback() YFiles
files.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback 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.

int **remove**(String **pathname**)

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_logicalName() YFiles
files→setLogicalName()`files.set_logicalName()`

Changes the logical name of the filesystem.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the filesystem.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

files→set_userData() YFiles
files→setUserData()`files.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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.

int **upload**(String **pathname**)

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.17. GenericSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_genericsensor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YGenericSensor = yoctolib.YGenericSensor;
php     require_once('yocto_genericsensor.php');
cpp     #include "yocto_genericsensor.h"
m       #import "yocto_genericsensor.h"
pas     uses yocto_genericsensor;
vb      yocto_genericsensor.vb
cs      yocto_genericsensor.cs
java    import com.yoctopuce.YoctoAPI.YGenericSensor;
py      from yocto_genericsensor import *
```

Global functions
yFindGenericSensor(**func**) Retrieves a generic sensor for a given identifier.
yFirstGenericSensor() Starts the enumeration of generic sensors currently accessible.
`YGenericSensor` methods
genericsensor→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
genericsensor→describe() Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
genericsensor→get_advertisedValue() Returns the current value of the generic sensor (no more than 6 characters).
genericsensor→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
genericsensor→get_currentValue() Returns the current measured value.
genericsensor→get_errorMessage() Returns the error message of the latest error with the generic sensor.
genericsensor→get_errorType() Returns the numerical error code of the latest error with the generic sensor.
genericsensor→get_friendlyName() Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.
genericsensor→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
genericsensor→get_functionId() Returns the hardware identifier of the generic sensor, without reference to the module.
genericsensor→get_hardwareId() Returns the unique hardware identifier of the generic sensor in the form `SERIAL.FUNCTIONID`.

genericsensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

genericsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

genericsensor→get_logicalName()

Returns the logical name of the generic sensor.

genericsensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

genericsensor→get_module()

Gets the `YModule` object for the device on which the function is located.

genericsensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

genericsensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

genericsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

genericsensor→get_resolution()

Returns the resolution of the measured values.

genericsensor→get_signalRange()

Returns the electric signal range used by the sensor.

genericsensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericsensor→get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

genericsensor→get_unit()

Returns the measuring unit for the measure.

genericsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

genericsensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericsensor→isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

genericsensor→isOnline_async(**callback**, **context**)

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

genericsensor→load(**msValidity**)

Preloads the generic sensor cache with a specified validity duration.

genericsensor→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

genericsensor→load_async(**msValidity**, **callback**, **context**)

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

genericsensor→nextGenericSensor()

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

genericsensor→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

genericsensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

genericsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

genericsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

genericsensor→set_logicalName(newval)

Changes the logical name of the generic sensor.

genericsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

genericsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

genericsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

genericsensor→set_signalRange(newval)

Changes the electric signal range used by the sensor.

genericsensor→set_unit(newval)

Changes the measuring unit for the measured value.

genericsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

genericsensor→set_valueRange(newval)

Changes the physical value range measured by the sensor.

genericsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGenericSensor.FindGenericSensor() YGenericSensor
yFindGenericSensor()
`YGenericSensor.FindGenericSensor()`

Retrieves a generic sensor for a given identifier.

YGenericSensor **FindGenericSensor**(String **func**)

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() YGenericSensor yFirstGenericSensor() YGenericSensor.FirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

```
YGenericSensor 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() YGenericSensor
genericsensor.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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() YGenericSensor
genericsensor.describe()

Returns a short text that describes unambiguously the instance of the generic sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 generic sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

genericsensor→get_advertisedValue() YGenericSensor
genericsensor→advertisedValue()
genericsensor.get_advertisedValue()

Returns the current value of the generic sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the generic sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

genericsensor→get_currentRawValue() YGenericSensor
genericsensor→currentRawValue()
genericsensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **get_currentRawValue**( )

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

genericsensor→get_currentValue() YGenericSensor
genericsensor→currentValue()
genericsensor.get_currentValue()

Returns the current measured value.

double **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() YGenericSensor
genericsensor→errorMessage()
genericsensor.get_errorMessage()

Returns the error message of the latest error with the generic sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the generic sensor object

genericsensor→get_errorType() YGenericSensor
genericsensor→errorType()
genericsensor.get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

int **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() YGenericSensor
genericsensor→friendlyName()
genericsensor.get_friendlyName()

Returns a global identifier of the generic sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YGenericSensor
genericsensor→functionDescriptor()
genericsensor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

genericsensor→get_functionId() YGenericSensor
genericsensor→functionId()
genericsensor.get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the generic sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

genericsensor→get_hardwareId() YGenericSensor
genericsensor→hardwareId()
genericsensor.get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the generic sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the generic sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

genericsensor→get_highestValue() YGenericSensor
genericsensor→highestValue()
genericsensor.get_highestValue()

Returns the maximal value observed for the measure since the device was started.

double **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

genericsensor→get_logFrequency() YGenericSensor
genericsensor→logFrequency()
genericsensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

genericsensor→get_logicalName() YGenericSensor
genericsensor→logicalName()
genericsensor.get_logicalName()

Returns the logical name of the generic sensor.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the generic sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

genericsensor→get_lowestValue() YGenericSensor
genericsensor→lowestValue()
genericsensor.get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

double **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

genericsensor→get_module() YGenericSensor
genericsensor→module()
genericsensor.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

genericsensor→get_recordedData() YGenericSensor
genericsensor→recordedData()
genericsensor.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

genericsensor→get_reportFrequency() YGenericSensor
genericsensor→reportFrequency()
genericsensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

genericsensor→get_resolution() YGenericSensor
genericsensor→resolution()
genericsensor.get_resolution()

Returns the resolution of the measured values.

double **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() YGenericSensor
genericsensor→signalRange()
genericsensor.get_signalRange()

Returns the electric signal range used by the sensor.

String **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() YGenericSensor
genericsensor→signalUnit()
genericsensor.get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

String **get_signalUnit**( )

Returns :
a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALUNIT_INVALID.

genericsensor→get_signalValue() YGenericSensor
genericsensor→signalValue()
genericsensor.get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

double **get_signalValue**( )

Returns :
a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALVALUE_INVALID.

genericsensor→get_unit() YGenericSensor
genericsensor→unit()genericsensor.get_unit()

Returns the measuring unit for the measure.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns Y_UNIT_INVALID.

genericsensor→get_userData() YGenericSensor
genericsensor→userData()
genericsensor.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

genericsensor→get_valueRange() YGenericSensor
genericsensor→valueRange()
genericsensor.get_valueRange()

Returns the physical value range measured by the sensor.

String **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() YGenericSensor
genericsensor.isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

```
boolean isOnline( )
```

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the generic sensor.

Returns :

`true` if the generic sensor can be reached, and `false` otherwise

genericsensor→load()genericsensor.load() YGenericSensor

Preloads the generic sensor cache with a specified validity duration.

int **load**(long **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() YGenericSensor
genericsensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→nextGenericSensor() YGenericSensor
genericsensor.nextGenericSensor()

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

YGenericSensor **nextGenericSensor**( )

Returns :

a pointer to a `YGenericSensor` object, corresponding to a generic sensor currently online, or a `null` pointer if there are no more generic sensors to enumerate.

genericsensor→registerTimedReportCallback() YGenericSensor
genericsensor.registerTimedReportCallback( )

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

genericsensor→registerValueCallback() YGenericSensor
genericsensor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

genericsensor→set_highestValue() YGenericSensor
genericsensor→setHighestValue()
genericsensor.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logFrequency() YGenericSensor
genericsensor→setLogFrequency()
genericsensor.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logicalName() YGenericSensor
genericsensor→setLogicalName()
genericsensor.set_logicalName()

Changes the logical name of the generic sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the generic sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

genericsensor→set_lowestValue() YGenericSensor
genericsensor→setLowestValue()
genericsensor.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_reportFrequency() YGenericSensor
genericsensor→setReportFrequency()
genericsensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_resolution() YGenericSensor
genericsensor→setResolution()
genericsensor.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **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() YGenericSensor
genericsensor→setSignalRange()
genericsensor.set_signalRange()

Changes the electric signal range used by the sensor.

int **set_signalRange**(String **newval**)

Parameters :

newval a string corresponding to the electric signal range used by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_unit() YGenericSensor
genericsensor→setUnit()
genericsensor.set_unit()

Changes the measuring unit for the measured value.

int **set_unit**(String **newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the measuring unit for the measured value

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_userData() YGenericSensor
genericsensor→setUserData()
genericsensor.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

genericsensor→set_valueRange() YGenericSensor
genericsensor→setValueRange()
genericsensor.set_valueRange()

Changes the physical value range measured by the sensor.

int **set_valueRange**(String **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.

3.18. Gyroscope function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_gyro.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YGyro = yoctolib.YGyro;
php     require_once('yocto_gyro.php');
cpp     #include "yocto_gyro.h"
m       #import "yocto_gyro.h"
pas     uses yocto_gyro;
vb      yocto_gyro.vb
cs      yocto_gyro.cs
java    import com.yoctopuce.YoctoAPI.YGyro;
py      from yocto_gyro import *
```

Global functions
yFindGyro(**func**) Retrieves a gyroscope for a given identifier.
yFirstGyro() Starts the enumeration of gyroscopes currently accessible.
YGyro methods
gyro→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
gyro→describe() Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
gyro→get_advertisedValue() Returns the current value of the gyroscope (no more than 6 characters).
gyro→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
gyro→get_currentValue() Returns the current value of the angular velocity.
gyro→get_errorMessage() Returns the error message of the latest error with the gyroscope.
gyro→get_errorType() Returns the numerical error code of the latest error with the gyroscope.
gyro→get_friendlyName() Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.
gyro→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
gyro→get_functionId() Returns the hardware identifier of the gyroscope, without reference to the module.
gyro→get_hardwareId() Returns the unique hardware identifier of the gyroscope in the form `SERIAL.FUNCTIONID`.

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

gyro→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

gyro→get_logicalName()

Returns the logical name of the gyroscope.

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

gyro→get_module()

Gets the `YModule` object for the device on which the function is located.

gyro→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionW()

Returns the `w` component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionX()

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionY()

Returns the `y` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionZ()

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

gyro→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

gyro→get_resolution()

Returns the resolution of the measured values.

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_unit()

Returns the measuring unit for the angular velocity.

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

gyro→get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

gyro→get_yValue()

Returns the angular velocity around the Y axis of the device, as a floating point number.

gyro→get_zValue()
Returns the angular velocity around the Z axis of the device, as a floating point number.

gyro→isOnline()
Checks if the gyroscope is currently reachable, without raising any error.

gyro→isOnline_async(**callback**, **context**)
Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

gyro→load(**msValidity**)
Preloads the gyroscope cache with a specified validity duration.

gyro→loadCalibrationPoints(**rawValues**, **refValues**)
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

gyro→load_async(**msValidity**, **callback**, **context**)
Preloads the gyroscope cache with a specified validity duration (asynchronous version).

gyro→nextGyro()
Continues the enumeration of gyroscopes started using `yFirstGyro()`.

gyro→registerAnglesCallback(**callback**)
Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerQuaternionCallback(**callback**)
Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerTimedReportCallback(**callback**)
Registers the callback function that is invoked on every periodic timed notification.

gyro→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

gyro→set_highestValue(**newval**)
Changes the recorded maximal value observed.

gyro→set_logFrequency(**newval**)
Changes the datalogger recording frequency for this function.

gyro→set_logicalName(**newval**)
Changes the logical name of the gyroscope.

gyro→set_lowestValue(**newval**)
Changes the recorded minimal value observed.

gyro→set_reportFrequency(**newval**)
Changes the timed value notification frequency for this function.

gyro→set_resolution(**newval**)
Changes the resolution of the measured physical values.

gyro→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

gyro→wait_async(**callback**, **context**)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGyro.FindGyro() YGyro yFindGyro()YGyro.FindGyro()

Retrieves a gyroscope for a given identifier.

YGyro **FindGyro**(String **func**)

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() YGyro yFirstGyro()YGyro.FirstGyro()

Starts the enumeration of gyroscopes currently accessible.

```
YGyro 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() YGyro
gyro.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                                                ArrayList<Double> 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.

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 gyroscope (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

gyro→get_advertisedValue() YGyro
gyro→advertisedValue()
gyro.get_advertisedValue()

Returns the current value of the gyroscope (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the gyroscope (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

gyro→get_currentRawValue() YGyro
gyro→currentRawValue()
gyro.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YGyro
gyro→currentValue()gyro.get_currentValue()

Returns the current value of the angular velocity.

double **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() YGyro
gyro→errorMessage()gyro.get_errorMessage()

Returns the error message of the latest error with the gyroscope.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the gyroscope object

gyro→get_errorType() YGyro
gyro→errorType()gyro.get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

int **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() YGyro
gyro→friendlyName()gyro.get_friendlyName()

Returns a global identifier of the gyroscope in the format MODULE_NAME.FUNCTION_NAME.

String **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() YGyro
gyro→functionDescriptor()
gyro.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

gyro→get_functionId() YGyro
gyro→functionId()gyro.get_functionId()

Returns the hardware identifier of the gyroscope, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the gyroscope (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

gyro→get_hardwareId() YGyro
gyro→hardwareId()gyro.get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the gyroscope. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the gyroscope (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

gyro→get_heading() YGyro
gyro→heading()gyro.get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_heading**( )

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class YRefFrame.

Returns :
a floating-point number corresponding to heading in degrees, between 0 and 360.

gyro→get_highestValue() YGyro
gyro→highestValue()gyro.get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

double **get_highestValue**( )

Returns :

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

gyro→get_logFrequency() YGyro
gyro→logFrequency()gyro.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

gyro→get_logicalName() YGyro
gyro→logicalName()gyro.get_logicalName()

Returns the logical name of the gyroscope.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the gyroscope. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

gyro→get_lowestValue() YGyro
gyro→lowestValue()gyro.get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

double **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

gyro→get_module() YGyro
gyro→module()gyro.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

gyro→get_pitch() YGyro
gyro→pitch()gyro.get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_pitch**( )

The axis corresponding to the pitch angle can be mapped to any of the device X, Y or Z physical directions using methods of the class YRefFrame.

Returns :
a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

gyro→get_quaternionW() YGyro
gyro→quaternionW()gyro.get_quaternionW()

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_quaternionW**( )

Returns :

a floating-point number corresponding to the w component of the quaternion.

gyro→get_quaternionX() YGyro
gyro→quaternionX()gyro.get_quaternionX()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_quaternionX**( )

The x component is mostly correlated with rotations on the roll axis.

Returns :

a floating-point number corresponding to the x component of the quaternion.

gyro→get_quaternionY() YGyro
gyro→quaternionY()gyro.get_quaternionY()

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_quaternionY**( )

The y component is mostly correlated with rotations on the pitch axis.

Returns :

a floating-point number corresponding to the y component of the quaternion.

gyro→get_quaternionZ() YGyro
gyro→quaternionZ()gyro.get_quaternionZ()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_quaternionZ**( )

The x component is mostly correlated with changes of heading.

Returns :

a floating-point number corresponding to the Z component of the quaternion.

gyro→get_recordedData() YGyro
gyro→recordedData()gyro.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

gyro→get_reportFrequency() YGyro
gyro→reportFrequency()
gyro.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

gyro→get_resolution() YGyro
gyro→resolution()gyro.get_resolution()

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

gyro→get_roll() YGyro
gyro→roll()gyro.get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_roll**( )

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class YRefFrame.

Returns :
a floating-point number corresponding to roll angle in degrees, between -180 and +180.

gyro→get_unit() YGyro
gyro→unit()gyro.get_unit()

Returns the measuring unit for the angular velocity.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns Y_UNIT_INVALID.

gyro→get_userData() YGyro
gyro→userData()gyro.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

gyro→get_xValue() YGyro
gyro→xValue()gyro.get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

double **get_xValue**( )

Returns :
a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

gyro→get_yValue() YGyro
gyro→yValue()gyro.get_yValue()

Returns the angular velocity around the Y axis of the device, as a floating point number.

double **get_yValue**( )

Returns :
a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns Y_YVALUE_INVALID.

gyro→get_zValue() YGyro
gyro→zValue()gyro.get_zValue()

Returns the angular velocity around the Z axis of the device, as a floating point number.

double **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.

boolean **isOnline**( )

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

Returns :

`true` if the gyroscope can be reached, and `false` otherwise

gyro→load()gyro.load() YGyro

Preloads the gyroscope cache with a specified validity duration.

int **load**(long **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() YGyro
gyro.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→nextGyro()gyro.nextGyro() YGyro

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

YGyro **nextGyro**( )

Returns :

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.

gyro→registerAnglesCallback() YGyro
gyro.registerAnglesCallback()

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

int **registerAnglesCallback**(YAnglesCallback **callback**)

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGyro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

gyro→registerQuaternionCallback() YGyro
gyro.registerQuaternionCallback()

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

int **registerQuaternionCallback**(YQuatCallback **callback**)

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

gyro→registerTimedReportCallback() YGyro
gyro.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

gyro→registerValueCallback() YGyro
gyro.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

gyro→set_highestValue() YGyro
gyro→setHighestValue()
gyro.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logFrequency() YGyro
gyro→setLogFrequency()
gyro.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logicalName() YGyro
gyro→setLogicalName()gyro.set_logicalName()

Changes the logical name of the gyroscope.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the gyroscope.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

gyro→set_lowestValue() YGyro
gyro→setLowestValue()gyro.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_reportFrequency() YGyro
gyro→setReportFrequency()
gyro.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_resolution() YGyro
gyro→setResolution()gyro.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_userData() YGyro
gyro→setUserData()gyro.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.19. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_hubport.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YHubPort = yoctolib.YHubPort;
php     require_once('yocto_hubport.php');
cpp     #include "yocto_hubport.h"
m       #import "yocto_hubport.h"
pas     uses yocto_hubport;
vb      yocto_hubport.vb
cs      yocto_hubport.cs
java    import com.yoctopuce.YoctoAPI.YHubPort;
py      from yocto_hubport import *
```

Global functions

yFindHubPort(**func**)

Retrieves a Yocto-hub port for a given identifier.

yFirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

`YHubPort` methods

hubport→describe()

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

hubport→get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

hubport→get_baudRate()

Returns the current baud rate used by this Yocto-hub port, in kbps.

hubport→get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

hubport→get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

hubport→get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

hubport→get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

hubport→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

hubport→get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

hubport→get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

hubport→get_logicalName()

Returns the logical name of the Yocto-hub port.

hubport→get_module()

Gets the `YModule` object for the device on which the function is located.

hubport→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

hubport→get_portState()

Returns the current state of the Yocto-hub port.

hubport→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

hubport→isOnline()

Checks if the Yocto-hub port is currently reachable, without raising any error.

hubport→isOnline_async(**callback**, **context**)

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

hubport→load(**msValidity**)

Preloads the Yocto-hub port cache with a specified validity duration.

hubport→load_async(**msValidity**, **callback**, **context**)

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

hubport→nextHubPort()

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

hubport→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

hubport→set_enabled(**newval**)

Changes the activation of the Yocto-hub port.

hubport→set_logicalName(**newval**)

Changes the logical name of the Yocto-hub port.

hubport→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

hubport→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHubPort.FindHubPort() YHubPort yFindHubPort()YHubPort.FindHubPort()

Retrieves a Yocto-hub port for a given identifier.

YHubPort **FindHubPort**(String **func**)

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() YHubPort yFirstHubPort()YHubPort.FirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

```
YHubPort 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.

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 Yocto-hub port (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

hubport→get_advertisedValue() YHubPort
hubport→advertisedValue()
hubport.get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the Yocto-hub port (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

hubport→get_baudRate() YHubPort
hubport→baudRate()hubport.get_baudRate()

Returns the current baud rate used by this Yocto-hub port, in kbps.

int **get_baudRate**( )

The default value is 1000 kbps, but a slower rate may be used if communication problems are encountered.

Returns :
an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns Y_BAUDRATE_INVALID.

hubport→get_enabled() YHubPort
hubport→enabled()hubport.get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

int **get_enabled**( )

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the Yocto-hub port is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

hubport→get_errorMessage() YHubPort
hubport→errorMessage()
hubport.get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the Yocto-hub port object

hubport→get_errorType() YHubPort
hubport→errorType()hubport.get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

int **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() YHubPort
hubport→friendlyName()
hubport.get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format MODULE_NAME.FUNCTION_NAME.

String **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() YHubPort
hubport→functionDescriptor()
hubport.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

hubport→get_functionId() YHubPort
hubport→functionId()hubport.get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the Yocto-hub port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

hubport→get_hardwareId() YHubPort
hubport→hardwareId()hubport.get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Yocto-hub port. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the Yocto-hub port (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

hubport→get_logicalName() YHubPort
hubport→logicalName()
hubport.get_logicalName()

Returns the logical name of the Yocto-hub port.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the Yocto-hub port. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

hubport→get_module() YHubPort
hubport→module()hubport.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

hubport→get_portState() YHubPort
hubport→portState()hubport.get_portState()

Returns the current state of the Yocto-hub port.

int **get_portState**( )

Returns :

a value among Y_PORTSTATE_OFF, Y_PORTSTATE_OVRLD, Y_PORTSTATE_ON, Y_PORTSTATE_RUN and Y_PORTSTATE_PROG corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns Y_PORTSTATE_INVALID.

hubport→get_userData() YHubPort
hubport→userData()hubport.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Yocto-hub port.

Returns :
`true` if the Yocto-hub port can be reached, and `false` otherwise

hubport→load()hubport.load() YHubPort

Preloads the Yocto-hub port cache with a specified validity duration.

int **load**(long **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→nextHubPort()hubport.nextHubPort() YHubPort

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

YHubPort **nextHubPort**( )

Returns :

a pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a `null` pointer if there are no more Yocto-hub ports to enumerate.

hubport→registerValueCallback() YHubPort
hubport.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

hubport→set_enabled() YHubPort
hubport→setEnabled()hubport.set_enabled()

Changes the activation of the Yocto-hub port.

int **set_enabled**(int **newval**)

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the activation of the Yocto-hub port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→set_logicalName() YHubPort
hubport→setLogicalName()
hubport.set_logicalName()

Changes the logical name of the Yocto-hub port.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Yocto-hub port.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

hubport→set_userData() YHubPort
hubport→setUserData()hubport.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.20. Humidity function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_humidity.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YHumidity = yoctolib.YHumidity;
php     require_once('yocto_humidity.php');
cpp     #include "yocto_humidity.h"
m       #import "yocto_humidity.h"
pas     uses yocto_humidity;
vb      yocto_humidity.vb
cs      yocto_humidity.cs
java    import com.yoctopuce.YoctoAPI.YHumidity;
py      from yocto_humidity import *
```

Global functions
yFindHumidity(**func**) Retrieves a humidity sensor for a given identifier.
yFirstHumidity() Starts the enumeration of humidity sensors currently accessible.
`YHumidity` methods
humidity→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
humidity→describe() Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
humidity→get_advertisedValue() Returns the current value of the humidity sensor (no more than 6 characters).
humidity→get_currentRawValue() Returns the unrounded and uncalibrated raw value returned by the sensor.
humidity→get_currentValue() Returns the current measure for the humidity.
humidity→get_errorMessage() Returns the error message of the latest error with the humidity sensor.
humidity→get_errorType() Returns the numerical error code of the latest error with the humidity sensor.
humidity→get_friendlyName() Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.
humidity→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
humidity→get_functionId() Returns the hardware identifier of the humidity sensor, without reference to the module.
humidity→get_hardwareId() Returns the unique hardware identifier of the humidity sensor in the form `SERIAL.FUNCTIONID`.

humidity→get_highestValue()

Returns the maximal value observed for the humidity.

humidity→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

Returns the minimal value observed for the humidity.

humidity→get_module()

Gets the `YModule` object for the device on which the function is located.

humidity→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

humidity→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

humidity→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

humidity→get_resolution()

Returns the resolution of the measured values.

humidity→get_unit()

Returns the measuring unit for the humidity.

humidity→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

humidity→isOnline()

Checks if the humidity sensor is currently reachable, without raising any error.

humidity→isOnline_async(**callback**, **context**)

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

humidity→load(**msValidity**)

Preloads the humidity sensor cache with a specified validity duration.

humidity→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

humidity→load_async(**msValidity**, **callback**, **context**)

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

humidity→nextHumidity()

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

humidity→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

humidity→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

humidity→set_highestValue(**newval**)

Changes the recorded maximal value observed for the humidity.

humidity→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

humidity→set_logicalName(**newval**)

Changes the logical name of the humidity sensor.

humidity→set_lowestValue(newval)

Changes the recorded minimal value observed for the humidity.

humidity→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

humidity→set_resolution(newval)

Changes the resolution of the measured physical values.

humidity→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

humidity→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHumidity.FindHumidity() YHumidity yFindHumidity()YHumidity.FindHumidity()

Retrieves a humidity sensor for a given identifier.

YHumidity **FindHumidity**(String **func**)

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() YHumidity yFirstHumidity()YHumidity.FirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

YHumidity **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() YHumidity
humidity.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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 humidity sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

humidity→get_advertisedValue() YHumidity
humidity→advertisedValue()
humidity.get_advertisedValue()

Returns the current value of the humidity sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the humidity sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

humidity→get_currentRawValue() YHumidity
humidity→currentRawValue()
humidity.get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **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() YHumidity
humidity→currentValue()
humidity.get_currentValue()

Returns the current measure for the humidity.

double **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() YHumidity
humidity→errorMessage()
humidity.get_errorMessage()

Returns the error message of the latest error with the humidity sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the humidity sensor object

humidity→get_errorType() YHumidity
humidity→errorType()humidity.get_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

int **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() YHumidity
humidity→friendlyName()
humidity.get_friendlyName()

Returns a global identifier of the humidity sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YHumidity
humidity→functionDescriptor()
humidity.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

humidity→get_functionId() YHumidity
humidity→functionId()
humidity.get_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the humidity sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

humidity→get_hardwareId() YHumidity
humidity→hardwareId()
humidity.get_hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the humidity sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the humidity sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

humidity→get_highestValue() YHumidity
humidity→highestValue()
humidity.get_highestValue()

Returns the maximal value observed for the humidity.

double **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() YHumidity
humidity→logFrequency()
humidity.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

humidity→get_logicalName() YHumidity
humidity→logicalName()
humidity.get_logicalName()

Returns the logical name of the humidity sensor.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the humidity sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

humidity→get_lowestValue() YHumidity
humidity→lowestValue()
humidity.get_lowestValue()

Returns the minimal value observed for the humidity.

double **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() YHumidity
humidity→module()humidity.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

humidity→get_recordedData() YHumidity
humidity→recordedData()
humidity.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **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() YHumidity
humidity→reportFrequency()
humidity.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

humidity→get_resolution() YHumidity
humidity→resolution()
humidity.get_resolution()

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

humidity→get_unit() YHumidity
humidity→unit()humidity.get_unit()

Returns the measuring unit for the humidity.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

humidity→get_userData() YHumidity
humidity→userData()humidity.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the humidity sensor.

Returns :

`true` if the humidity sensor can be reached, and `false` otherwise

humidity→load()humidity.load() YHumidity

Preloads the humidity sensor cache with a specified validity duration.

int **load**(long **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() YHumidity
humidity.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→nextHumidity() YHumidity
humidity.nextHumidity()

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

YHumidity **nextHumidity**( )

Returns :

a pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a `null` pointer if there are no more humidity sensors to enumerate.

humidity→registerTimedReportCallback() YHumidity
humidity.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

humidity→registerValueCallback() YHumidity
humidity.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

humidity→set_highestValue() YHumidity
humidity→setHighestValue()
humidity.set_highestValue()

Changes the recorded maximal value observed for the humidity.

int **set_highestValue**(double **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() YHumidity
humidity→setLogFrequency()
humidity.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logicalName() YHumidity
humidity→setLogicalName()
humidity.set_logicalName()

Changes the logical name of the humidity sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the humidity sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

humidity→set_lowestValue() YHumidity
humidity→setLowestValue()
humidity.set_lowestValue()

Changes the recorded minimal value observed for the humidity.

int **set_lowestValue**(double **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() YHumidity
humidity→setReportFrequency()
humidity.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_resolution() YHumidity
humidity→setResolution()
humidity.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_userData() YHumidity
humidity→setUserData()
humidity.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.21. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_led.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YLed = yoctolib.YLed;
php     require_once('yocto_led.php');
cpp     #include "yocto_led.h"
m       #import "yocto_led.h"
pas     uses yocto_led;
vb      yocto_led.vb
cs      yocto_led.cs
java    import com.yoctopuce.YoctoAPI.YLed;
py      from yocto_led import *
```

Global functions
yFindLed(**func**) Retrieves a led for a given identifier.
yFirstLed() Starts the enumeration of leds currently accessible.
YLed methods
led→describe() Returns a short text that describes unambiguously the instance of the led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
led→get_advertisedValue() Returns the current value of the led (no more than 6 characters).
led→get_blinking() Returns the current led signaling mode.
led→get_errorMessage() Returns the error message of the latest error with the led.
led→get_errorType() Returns the numerical error code of the latest error with the led.
led→get_friendlyName() Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`.
led→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
led→get_functionId() Returns the hardware identifier of the led, without reference to the module.
led→get_hardwareId() Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`.
led→get_logicalName() Returns the logical name of the led.
led→get_luminosity() Returns the current led intensity (in per cent).
led→get_module()

Gets the YModule object for the device on which the function is located.

led→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

led→get_power()

Returns the current led state.

led→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

led→isOnline()

Checks if the led is currently reachable, without raising any error.

led→isOnline_async(callback, context)

Checks if the led is currently reachable, without raising any error (asynchronous version).

led→load(msValidity)

Preloads the led cache with a specified validity duration.

led→load_async(msValidity, callback, context)

Preloads the led cache with a specified validity duration (asynchronous version).

led→nextLed()

Continues the enumeration of leds started using yFirstLed().

led→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

led→set_blinking(newval)

Changes the current led signaling mode.

led→set_logicalName(newval)

Changes the logical name of the led.

led→set_luminosity(newval)

Changes the current led intensity (in per cent).

led→set_power(newval)

Changes the state of the led.

led→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

led→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLed.FindLed() YLed yFindLed()YLed.FindLed()

Retrieves a led for a given identifier.

```
YLed FindLed( String func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the led is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLed.isOnline()` to test if the led is indeed online at a given time. In case of ambiguity when looking for a led by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the led

Returns :

a `YLed` object allowing you to drive the led.

YLed.FirstLed() YLed yFirstLed()YLed.FirstLed()

Starts the enumeration of leds currently accessible.

```
YLed 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`.

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 led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

led→get_advertisedValue() YLed
led→advertisedValue()
`led.get_advertisedValue()`

Returns the current value of the led (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the led (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

led→get_blinking() YLed
led→blinking()`led.get_blinking()`

Returns the current led signaling mode.

int **get_blinking**( )

Returns :

a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

On failure, throws an exception or returns `Y_BLINKING_INVALID`.

led→get_errorMessage() YLed
led→errorMessage()`led.get_errorMessage()`

Returns the error message of the latest error with the led.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the led object

led→get_errorType() YLed
led→errorType()led.get_errorType()

Returns the numerical error code of the latest error with the led.

int **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() YLed
led→friendlyName()`led.get_friendlyName()`

Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`.

String **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() YLed
led→functionDescriptor()
led.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

led→get_functionId() YLed
led→functionId()`led.get_functionId()`

Returns the hardware identifier of the led, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the led (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

led→get_hardwareId() YLed
led→hardwareId()`led.get_hardwareId()`

Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the led. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the led (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

led→get_logicalName() YLed
led→logicalName()led.get_logicalName()

Returns the logical name of the led.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the led. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

led→get_luminosity() YLed
led→luminosity()led.get_luminosity()

Returns the current led intensity (in per cent).

int **get_luminosity**( )

Returns :
an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.

led→get_module() YLed
led→module()led.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

led→get_power() YLed
led→power()led.get_power()

Returns the current led state.

int **get_power**( )

Returns :

either Y_POWER_OFF or Y_POWER_ON, according to the current led state

On failure, throws an exception or returns Y_POWER_INVALID.

led→get_userData() YLed
led→userData()`led.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the led.

Returns :
`true` if the led can be reached, and `false` otherwise

led→load()`led.load()` YLed

Preloads the led cache with a specified validity duration.

int **load**(long **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→nextLed()`led.nextLed()` YLed

Continues the enumeration of leds started using `yFirstLed()`.

YLed **nextLed**( )

Returns :

a pointer to a `YLed` object, corresponding to a led currently online, or a `null` pointer if there are no more leds to enumerate.

led→registerValueCallback() YLed
led.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

led→set_blinking() YLed
led→setBlinking()led.set_blinking()

Changes the current led signaling mode.

int **set_blinking**(int **newval**)

Parameters :

newval a value among Y_BLINKING_STILL, Y_BLINKING_RELAX, Y_BLINKING_AWARE, Y_BLINKING_RUN, Y_BLINKING_CALL and Y_BLINKING_PANIC corresponding to the current led signaling mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→set_logicalName() YLed
led→setLogicalName()led.set_logicalName()

Changes the logical name of the led.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the led.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

led→set_luminosity() YLed
led→setLuminosity()led.set_luminosity()

Changes the current led intensity (in per cent).

int **set_luminosity**(int **newval**)

Parameters :

newval an integer corresponding to the current led intensity (in per cent)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→set_power() YLed
led→setPower()led.set_power()

Changes the state of the led.

int **set_power**(int **newval**)

Parameters :

newval either Y_POWER_OFF or Y_POWER_ON, according to the state of the led

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→set_userData() YLed
led→setUserData()`led.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.22. LightSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_lightsensor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YLightSensor = yoctolib.YLightSensor;
php     require_once('yocto_lightsensor.php');
cpp     #include "yocto_lightsensor.h"
m       #import "yocto_lightsensor.h"
pas     uses yocto_lightsensor;
vb      yocto_lightsensor.vb
cs      yocto_lightsensor.cs
java    import com.yoctopuce.YoctoAPI.YLightSensor;
py      from yocto_lightsensor import *
```

Global functions
yFindLightSensor(**func**) Retrieves a light sensor for a given identifier.
yFirstLightSensor() Starts the enumeration of light sensors currently accessible.
`YLightSensor` methods
lightsensor→calibrate(**calibratedVal**) Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).
lightsensor→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
lightsensor→describe() Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
lightsensor→get_advertisedValue() Returns the current value of the light sensor (no more than 6 characters).
lightsensor→get_currentRawValue() Returns the unrounded and uncalibrated raw value returned by the sensor.
lightsensor→get_currentValue() Returns the current measure for the ambiant light.
lightsensor→get_errorMessage() Returns the error message of the latest error with the light sensor.
lightsensor→get_errorType() Returns the numerical error code of the latest error with the light sensor.
lightsensor→get_friendlyName() Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.
lightsensor→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
lightsensor→get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

lightsensor→get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

lightsensor→get_highestValue()

Returns the maximal value observed for the ambiant light.

lightsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

lightsensor→get_logicalName()

Returns the logical name of the light sensor.

lightsensor→get_lowestValue()

Returns the minimal value observed for the ambiant light.

lightsensor→get_module()

Gets the `YModule` object for the device on which the function is located.

lightsensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

lightsensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

lightsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

lightsensor→get_resolution()

Returns the resolution of the measured values.

lightsensor→get_unit()

Returns the measuring unit for the ambiant light.

lightsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

lightsensor→isOnline()

Checks if the light sensor is currently reachable, without raising any error.

lightsensor→isOnline_async(**callback**, **context**)

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

lightsensor→load(**msValidity**)

Preloads the light sensor cache with a specified validity duration.

lightsensor→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

lightsensor→load_async(**msValidity**, **callback**, **context**)

Preloads the light sensor cache with a specified validity duration (asynchronous version).

lightsensor→nextLightSensor()

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

lightsensor→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

lightsensor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

lightsensor→set_highestValue(**newval**)

Changes the recorded maximal value observed for the ambiant light.

lightsensor→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

lightsensor→set_logicalName(newval)

Changes the logical name of the light sensor.

lightsensor→set_lowestValue(newval)

Changes the recorded minimal value observed for the ambiant light.

lightsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

lightsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

lightsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

lightsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLightSensor.FindLightSensor() YLightSensor
yFindLightSensor()
YLightSensor.FindLightSensor()

Retrieves a light sensor for a given identifier.

YLightSensor **FindLightSensor**(String **func**)

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() YLightSensor yFirstLightSensor() YLightSensor.FirstLightSensor()

Starts the enumeration of light sensors currently accessible.

```
YLightSensor 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).

int **calibrate**(double **calibratedVal**)

Parameters :

calibratedVal the desired target value.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→calibrateFromPoints() YLightSensor
lightsensor.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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`.

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 light sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

lightsensor→get_advertisedValue() YLightSensor
lightsensor→advertisedValue()
lightsensor.get_advertisedValue()

Returns the current value of the light sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the light sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

lightsensor→get_currentRawValue() YLightSensor
lightsensor→currentRawValue()
lightsensor.get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **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() YLightSensor
lightsensor→currentValue()
`lightsensor.get_currentValue()`

Returns the current measure for the ambiant light.

double **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() YLightSensor
lightsensor→errorMessage()
lightsensor.get_errorMessage()

Returns the error message of the latest error with the light sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the light sensor object

lightsensor→get_errorType() YLightSensor
lightsensor→errorType()
`lightsensor.get_errorType()`

Returns the numerical error code of the latest error with the light sensor.

```
int 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() YLightSensor
lightsensor→friendlyName()
lightsensor.get_friendlyName()

Returns a global identifier of the light sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YLightSensor
lightsensor→functionDescriptor()
`lightsensor.get_functionDescriptor()`

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

lightsensor→get_functionId() YLightSensor
lightsensor→functionId()
lightsensor.get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the light sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

lightsensor→get_hardwareId() YLightSensor
lightsensor→hardwareId()
lightsensor.get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the light sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the light sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

lightsensor→get_highestValue() YLightSensor
lightsensor→highestValue()
lightsensor.get_highestValue()

Returns the maximal value observed for the ambiant light.

double **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() YLightSensor
lightsensor→logFrequency()
`lightsensor.get_logFrequency()`

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

lightsensor→get_logicalName() YLightSensor
lightsensor→logicalName()
lightsensor.get_logicalName()

Returns the logical name of the light sensor.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the light sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

lightsensor→get_lowestValue() YLightSensor
lightsensor→lowestValue()
`lightsensor.get_lowestValue()`

Returns the minimal value observed for the ambiant light.

double **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() YLightSensor
lightsensor→module()`lightsensor.get_module()`

Gets the `YModule` object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :
an instance of `YModule`

lightsensor→get_recordedData() YLightSensor
lightsensor→recordedData()
`lightsensor.get_recordedData()`

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

lightsensor→get_reportFrequency() YLightSensor
lightsensor→reportFrequency()
`lightsensor.get_reportFrequency()`

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

lightsensor→get_resolution() YLightSensor
lightsensor→resolution()
`lightsensor.get_resolution()`

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

lightsensor→get_unit() YLightSensor
lightsensor→unit()`lightsensor.get_unit()`

Returns the measuring unit for the ambiant light.

String **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() YLightSensor
lightsensor→userData()
`lightsensor.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

Returns :
`true` if the light sensor can be reached, and `false` otherwise

lightsensor→load()`lightsensor.load()` YLightSensor

Preloads the light sensor cache with a specified validity duration.

int **load**(long **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() YLightSensor
lightsensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

int **loadCalibrationPoints**(ArrayList<Double> **rawValues**,
ArrayList<Double> **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→nextLightSensor() YLightSensor
lightsensor.nextLightSensor()

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

YLightSensor **nextLightSensor**( )

Returns :

a pointer to a `YLightSensor` object, corresponding to a light sensor currently online, or a `null` pointer if there are no more light sensors to enumerate.

lightsensor→registerTimedReportCallback() YLightSensor
`lightsensor.registerTimedReportCallback()`

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

lightsensor→registerValueCallback() YLightSensor
lightsensor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

lightsensor→set_highestValue() YLightSensor
lightsensor→setHighestValue()
lightsensor.set_highestValue()

Changes the recorded maximal value observed for the ambiant light.

int **set_highestValue**(double **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() YLightSensor
lightsensor→setLogFrequency()
`lightsensor.set_logFrequency()`

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logicalName() YLightSensor
lightsensor→setLogicalName()
lightsensor.set_logicalName()

Changes the logical name of the light sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the light sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

lightsensor→set_lowestValue() YLightSensor
lightsensor→setLowestValue()
`lightsensor.set_lowestValue()`

Changes the recorded minimal value observed for the ambiant light.

int **set_lowestValue**(double **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() YLightSensor
lightsensor→setReportFrequency()
lightsensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_resolution() YLightSensor
lightsensor→setResolution()
`lightsensor.set_resolution()`

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_userData() YLightSensor
lightsensor→setUserData()
lightsensor.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.23. Magnetometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_magnetometer.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YMagnetometer = yoctolib.YMagnetometer;
php     require_once('yocto_magnetometer.php');
cpp     #include "yocto_magnetometer.h"
m       #import "yocto_magnetometer.h"
pas     uses yocto_magnetometer;
vb      yocto_magnetometer.vb
cs      yocto_magnetometer.cs
java    import com.yoctopuce.YoctoAPI.YMagnetometer;
py      from yocto_magnetometer import *
```

Global functions
yFindMagnetometer(**func**) Retrieves a magnetometer for a given identifier.
yFirstMagnetometer() Starts the enumeration of magnetometers currently accessible.
`YMagnetometer` methods
magnetometer→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
magnetometer→describe() Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
magnetometer→get_advertisedValue() Returns the current value of the magnetometer (no more than 6 characters).
magnetometer→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
magnetometer→get_currentValue() Returns the current value of the magnetic field.
magnetometer→get_errorMessage() Returns the error message of the latest error with the magnetometer.
magnetometer→get_errorType() Returns the numerical error code of the latest error with the magnetometer.
magnetometer→get_friendlyName() Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.
magnetometer→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
magnetometer→get_functionId() Returns the hardware identifier of the magnetometer, without reference to the module.
magnetometer→get_hardwareId() Returns the unique hardware identifier of the magnetometer in the form `SERIAL.FUNCTIONID`.

magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

magnetometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

magnetometer→get_logicalName()

Returns the logical name of the magnetometer.

magnetometer→get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

magnetometer→get_module()

Gets the `YModule` object for the device on which the function is located.

magnetometer→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

magnetometer→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

magnetometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

magnetometer→get_resolution()

Returns the resolution of the measured values.

magnetometer→get_unit()

Returns the measuring unit for the magnetic field.

magnetometer→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

magnetometer→get_xValue()

Returns the X component of the magnetic field, as a floating point number.

magnetometer→get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

magnetometer→get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

magnetometer→isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

magnetometer→isOnline_async(callback, context)

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

magnetometer→load(msValidity)

Preloads the magnetometer cache with a specified validity duration.

magnetometer→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

magnetometer→load_async(msValidity, callback, context)

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

magnetometer→nextMagnetometer()

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

magnetometer→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

magnetometer→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

magnetometer→set_highestValue(newval)

Changes the recorded maximal value observed.

magnetometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

magnetometer→set_logicalName(newval)

Changes the logical name of the magnetometer.

magnetometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

magnetometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

magnetometer→set_resolution(newval)

Changes the resolution of the measured physical values.

magnetometer→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

magnetometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMagnetometer.FindMagnetometer() YMagnetometer
yFindMagnetometer()
YMagnetometer.FindMagnetometer()

Retrieves a magnetometer for a given identifier.

YMagnetometer **FindMagnetometer**(String **func**)

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() YMagnetometer yFirstMagnetometer() YMagnetometer.FirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

YMagnetometer **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() YMagnetometer
magnetometer.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                                                    ArrayList<Double> 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() YMagnetometer
magnetometer.describe()

Returns a short text that describes unambiguously the instance of the magnetometer in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 magnetometer (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

magnetometer→get_advertisedValue() YMagnetometer
magnetometer→advertisedValue()
magnetometer.get_advertisedValue()

Returns the current value of the magnetometer (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the magnetometer (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

magnetometer→get_currentRawValue() YMagnetometer
magnetometer→currentRawValue()
magnetometer.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YMagnetometer
magnetometer→currentValue()
magnetometer.get_currentValue()

Returns the current value of the magnetic field.

double **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() YMagnetometer
magnetometer→errorMessage()
magnetometer.get_errorMessage()

Returns the error message of the latest error with the magnetometer.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the magnetometer object

magnetometer→get_errorType() YMagnetometer
magnetometer→errorType()
magnetometer.get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

int **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() YMagnetometer
magnetometer→friendlyName()
magnetometer.get_friendlyName()

Returns a global identifier of the magnetometer in the format MODULE_NAME.FUNCTION_NAME.

String **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() YMagnetometer
magnetometer→functionDescriptor()
magnetometer.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

magnetometer→get_functionId() YMagnetometer
magnetometer→functionId()
magnetometer.get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the magnetometer (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

magnetometer→get_hardwareId() YMagnetometer
magnetometer→hardwareId()
magnetometer.get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the magnetometer. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the magnetometer (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

magnetometer→get_highestValue() YMagnetometer
magnetometer→highestValue()
magnetometer.get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

double **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

magnetometer→get_logFrequency() YMagnetometer
magnetometer→logFrequency()
magnetometer.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

magnetometer→get_logicalName() YMagnetometer
magnetometer→logicalName()
magnetometer.get_logicalName()

Returns the logical name of the magnetometer.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the magnetometer. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

magnetometer→get_lowestValue() YMagnetometer
magnetometer→lowestValue()
magnetometer.get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

double **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

magnetometer→get_module() YMagnetometer
magnetometer→module()
magnetometer.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

magnetometer→get_recordedData() YMagnetometer
magnetometer→recordedData()
magnetometer.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

magnetometer→get_reportFrequency() YMagnetometer
magnetometer→reportFrequency()
magnetometer.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

magnetometer→get_resolution() YMagnetometer
magnetometer→resolution()
magnetometer.get_resolution()

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

magnetometer→get_unit() YMagnetometer
magnetometer→unit()`magnetometer.get_unit()`

Returns the measuring unit for the magnetic field.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns `Y_UNIT_INVALID`.

magnetometer→get_userData() YMagnetometer
magnetometer→userData()
magnetometer.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

magnetometer→get_xValue() YMagnetometer
magnetometer→xValue()
magnetometer.get_xValue()

Returns the X component of the magnetic field, as a floating point number.

double **get_xValue**( )

Returns :
a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

magnetometer→get_yValue() YMagnetometer
magnetometer→yValue()
magnetometer.get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

double **get_yValue**( )

Returns :
a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_YVALUE_INVALID.

magnetometer→get_zValue() YMagnetometer
magnetometer→zValue()
magnetometer.get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

double **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() YMagnetometer
magnetometer.isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

boolean **isOnline**( )

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the magnetometer.

Returns :

`true` if the magnetometer can be reached, and `false` otherwise

magnetometer→load()`magnetometer.load()` YMagnetometer

Preloads the magnetometer cache with a specified validity duration.

int **load**(long **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() YMagnetometer
magnetometer.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

int **loadCalibrationPoints**(ArrayList<Double> **rawValues**,
ArrayList<Double> **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→nextMagnetometer() YMagnetometer
magnetometer.nextMagnetometer()

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

YMagnetometer **nextMagnetometer**( )

Returns :

a pointer to a `YMagnetometer` object, corresponding to a magnetometer currently online, or a `null` pointer if there are no more magnetometers to enumerate.

magnetometer→registerTimedReportCallback() YMagnetometer
magnetometer.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

magnetometer→registerValueCallback() YMagnetometer
magnetometer.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

magnetometer→set_highestValue() YMagnetometer
magnetometer→setHighestValue()
magnetometer.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logFrequency() YMagnetometer
magnetometer→setLogFrequency()
magnetometer.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logicalName() YMagnetometer
magnetometer→setLogicalName()
magnetometer.set_logicalName()

Changes the logical name of the magnetometer.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the magnetometer.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

magnetometer→set_lowestValue() YMagnetometer
magnetometer→setLowestValue()
magnetometer.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_reportFrequency() YMagnetometer
magnetometer→setReportFrequency()
magnetometer.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_resolution() YMagnetometer
magnetometer→setResolution()
magnetometer.set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_userData() YMagnetometer
magnetometer→setUserData()
magnetometer.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.24. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
```

YMeasure methods
measure→get_averageValue() Returns the average value observed during the time interval covered by this measure.
measure→get_endTimeUTC() Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).
measure→get_maxValue() Returns the largest value observed during the time interval covered by this measure.
measure→get_minValue() Returns the smallest value observed during the time interval covered by this measure.
measure→get_startTimeUTC() Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_averageValue() YMeasure
measure→averageValue()
measure.get_averageValue()

Returns the average value observed during the time interval covered by this measure.

double **get_averageValue**( )

Returns :
a floating-point number corresponding to the average value observed.

measure→get_endTimeUTC() YMeasure
measure→endTimeUTC()
measure.get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

double **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() YMeasure
measure→maxValue()`measure.get_maxValue()`

Returns the largest value observed during the time interval covered by this measure.

double **get_maxValue**( )

Returns :
a floating-point number corresponding to the largest value observed.

measure→get_minValue() YMeasure
measure→minValue()`measure.get_minValue()`

Returns the smallest value observed during the time interval covered by this measure.

double **get_minValue**( )

Returns :
a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC() YMeasure
measure→startTimeUTC()
`measure.get_startTimeUTC()`

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

double **get_startTimeUTC**( )

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :
an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

3.25. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
```

Global functions
yFindModule(**func**) Allows you to find a module from its serial number or from its logical name.
yFirstModule() Starts the enumeration of modules currently accessible.
`YModule` methods
module→describe() Returns a descriptive text that identifies the module.
module→download(**pathname**) Downloads the specified built-in file and returns a binary buffer with its content.
module→functionCount() Returns the number of functions (beside the "module" interface) available on the module.
module→functionId(**functionIndex**) Retrieves the hardware identifier of the *n*th function on the module.
module→functionName(**functionIndex**) Retrieves the logical name of the *n*th function on the module.
module→functionValue(**functionIndex**) Retrieves the advertised value of the *n*th function on the module.
module→get_beacon() Returns the state of the localization beacon.
module→get_errorMessage() Returns the error message of the latest error with this module object.
module→get_errorType() Returns the numerical error code of the latest error with this module object.
module→get_firmwareRelease() Returns the version of the firmware embedded in the module.
module→get_hardwareId() Returns the unique hardware identifier of the module.
module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()

Returns a string with last logs of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

Returns the luminosity of the module informative leds (from 0 to 100).

module→get_persistentSettings()

Returns the current state of persistent module settings.

module→get_productId()

Returns the USB device identifier of the module.

module→get_productName()

Returns the commercial name of the module, as set by the factory.

module→get_productRelease()

Returns the hardware release version of the module.

module→get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()

Returns the serial number of the module, as set by the factory.

module→get_upTime()

Returns the number of milliseconds spent since the module was powered on.

module→get_usbBandwidth()

Returns the number of USB interfaces used by the module.

module→get_usbCurrent()

Returns the current consumed by the module on the USB bus, in milli-amps.

module→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

module→isOnline()

Checks if the module is currently reachable, without raising any error.

module→isOnline_async(**callback**, **context**)

Checks if the module is currently reachable, without raising any error.

module→load(**msValidity**)

Preloads the module cache with a specified validity duration.

module→load_async(**msValidity**, **callback**, **context**)

Preloads the module cache with a specified validity duration (asynchronous version).

module→nextModule()

Continues the module enumeration started using `yFirstModule()`.

module→reboot(**secBeforeReboot**)

Schedules a simple module reboot after the given number of seconds.

module→registerLogCallback(**callback**)

todo

module→revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_usbBandwidth(newval)

Changes the number of USB interfaces used by the module.

module→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YModule.FindModule() YModule
yFindModule()YModule.FindModule()

Allows you to find a module from its serial number or from its logical name.

YModule **FindModule**(String **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() YModule yFirstModule()YModule.FirstModule()

Starts the enumeration of modules currently accessible.

```
YModule 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.

String **describe**( )

The text may include either the logical name or the serial number of the module.

Returns :
a string that describes the module

module→get_beacon() YModule
module→beacon()module.get_beacon()

Returns the state of the localization beacon.

```
int get_beacon( )
```

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module→get_errorMessage() YModule
module→errorMessage()
module.get_errorMessage()

Returns the error message of the latest error with this module object.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using this module object

module→get_errorType() YModule
module→errorType()module.get_errorType()

Returns the numerical error code of the latest error with this module object.

```
int 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() YModule
module→firmwareRelease()
module.get_firmwareRelease()

Returns the version of the firmware embedded in the module.

String **get_firmwareRelease**( )

Returns :
a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

module→get_hardwareId() YModule
module→hardwareId()module.get_hardwareId()

Returns the unique hardware identifier of the module.

String **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_lastLogs() YModule
module→lastLogs()`module.get_lastLogs()`

Returns a string with last logs of the module.

String **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() YModule
module→logicalName()
module.get_logicalName()

Returns the logical name of the module.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the module

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

module→get_luminosity() YModule
module→luminosity()`module.get_luminosity()`

Returns the luminosity of the module informative leds (from 0 to 100).

int **get_luminosity**( )

Returns :
an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

module→get_persistentSettings() YModule
module→persistentSettings()
module.get_persistentSettings()

Returns the current state of persistent module settings.

int **get_persistentSettings**( )

Returns :

a value among `Y_PERSISTENTSETTINGS_LOADED`, `Y_PERSISTENTSETTINGS_SAVED` and `Y_PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `Y_PERSISTENTSETTINGS_INVALID`.

module→get_productId() YModule
module→productId()`module.get_productId()`

Returns the USB device identifier of the module.

int **get_productId**( )

Returns :
an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns `Y_PRODUCTID_INVALID`.

module→get_productName() YModule
module→productName()
module.get_productName()

Returns the commercial name of the module, as set by the factory.

String **get_productName**( )

Returns :
a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `Y_PRODUCTNAME_INVALID`.

module→get_productRelease() YModule
module→productRelease()
module.get_productRelease()

Returns the hardware release version of the module.

int **get_productRelease**( )

Returns :
an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.

module→get_rebootCountdown() YModule
module→rebootCountdown()
module.get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

int **get_rebootCountdown**( )

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns Y_REBOOTCOUNTDOWN_INVALID.

module→get_serialNumber() YModule
module→serialNumber()
module.get_serialNumber()

Returns the serial number of the module, as set by the factory.

String **get_serialNumber**( )

Returns :
a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns Y_SERIALNUMBER_INVALID.

module→get_upTime() YModule
module→upTime()module.get_upTime()

Returns the number of milliseconds spent since the module was powered on.

long **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() YModule
module→usbBandwidth()
module.get_usbBandwidth()

Returns the number of USB interfaces used by the module.

int **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() YModule
module→usbCurrent()`module.get_usbCurrent()`

Returns the current consumed by the module on the USB bus, in milli-amps.

int **get_usbCurrent**( )

Returns :
an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

module→get_userData() YModule
module→userData()module.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

`true` if the module can be reached, and `false` otherwise

module→load()module.load() YModule

Preloads the module cache with a specified validity duration.

int **load**(long **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→nextModule()`module.nextModule()` YModule

Continues the module enumeration started using `yFirstModule()`.

YModule **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.

int **reboot**(int **secBeforeReboot**)

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→revertFromFlash() YModule
module.revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

int **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.

int **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() YModule
module→setBeacon()`module.set_beacon()`

Turns on or off the module localization beacon.

int **set_beacon**(int **newval**)

Parameters :

newval either `Y_BEACON_OFF` or `Y_BEACON_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_logicalName() YModule
module→setLogicalName()
module.set_logicalName()

Changes the logical name of the module.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_luminosity() YModule
module→setLuminosity()
module.set_luminosity()

Changes the luminosity of the module informative leds.

```
int set_luminosity( int 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() YModule
module→setUsbBandwidth()
module.set_usbBandwidth()

Changes the number of USB interfaces used by the module.

int **set_usbBandwidth**(int **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() YModule
module→setUserData()`module.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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() YModule
module.triggerFirmwareUpdate()

Schedules a module reboot into special firmware update mode.

int **triggerFirmwareUpdate**(int **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.

3.26. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_network.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YNetwork = yoctolib.YNetwork;
php     require_once('yocto_network.php');
cpp     #include "yocto_network.h"
m       #import "yocto_network.h"
pas     uses yocto_network;
vb      yocto_network.vb
cs      yocto_network.cs
java    import com.yoctopuce.YoctoAPI.YNetwork;
py      from yocto_network import *
```

Global functions
yFindNetwork(**func**) Retrieves a network interface for a given identifier.
yFirstNetwork() Starts the enumeration of network interfaces currently accessible.
`YNetwork` methods
network→callbackLogin(**username**, **password**) Connects to the notification callback and saves the credentials required to log into it.
network→describe() Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
network→get_adminPassword() Returns a hash string if a password has been set for user "admin", or an empty string otherwise.
network→get_advertisedValue() Returns the current value of the network interface (no more than 6 characters).
network→get_callbackCredentials() Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.
network→get_callbackEncoding() Returns the encoding standard to use for representing notification values.
network→get_callbackMaxDelay() Returns the maximum waiting time between two callback notifications, in seconds.
network→get_callbackMethod() Returns the HTTP method used to notify callbacks for significant state changes.
network→get_callbackMinDelay() Returns the minimum waiting time between two callback notifications, in seconds.
network→get_callbackUrl() Returns the callback URL to notify of significant state changes.
network→get_discoverable() Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→get_errorMessage()

Returns the error message of the latest error with the network interface.

network→get_errorType()

Returns the numerical error code of the latest error with the network interface.

network→get_friendlyName()

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

network→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

network→get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

network→get_hardwareId()

Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

network→get_ipAddress()

Returns the IP address currently in use by the device.

network→get_logicalName()

Returns the logical name of the network interface.

network→get_macAddress()

Returns the MAC address of the network interface.

network→get_module()

Gets the `YModule` object for the device on which the function is located.

network→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

network→get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

network→get_primaryDNS()

Returns the IP address of the primary name server to be used by the module.

network→get_readiness()

Returns the current established working mode of the network interface.

network→get_router()

Returns the IP address of the router on the device subnet (default gateway).

network→get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

network→get_subnetMask()

Returns the subnet mask currently used by the device.

network→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

network→get_userPassword()

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

network→get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→isOnline()

Checks if the network interface is currently reachable, without raising any error.

network→isOnline_async(**callback**, **context**)

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

network→load(msValidity)

Preloads the network interface cache with a specified validity duration.

network→load_async(msValidity, callback, context)

Preloads the network interface cache with a specified validity duration (asynchronous version).

network→nextNetwork()

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

network→ping(host)

Pings str_host to test the network connectivity.

network→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

network→set_adminPassword(newval)

Changes the password for the "admin" user.

network→set_callbackCredentials(newval)

Changes the credentials required to connect to the callback address.

network→set_callbackEncoding(newval)

Changes the encoding standard to use for representing notification values.

network→set_callbackMaxDelay(newval)

Changes the maximum waiting time between two callback notifications, in seconds.

network→set_callbackMethod(newval)

Changes the HTTP method used to notify callbacks for significant state changes.

network→set_callbackMinDelay(newval)

Changes the minimum waiting time between two callback notifications, in seconds.

network→set_callbackUrl(newval)

Changes the callback URL to notify significant state changes.

network→set_discoverable(newval)

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→set_logicalName(newval)

Changes the logical name of the network interface.

network→set_primaryDNS(newval)

Changes the IP address of the primary name server to be used by the module.

network→set_secondaryDNS(newval)

Changes the IP address of the secondary name server to be used by the module.

network→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

network→set_userPassword(newval)

Changes the password for the "user" user.

network→set_wwwWatchdogDelay(newval)

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

network→useStaticIP(ipAddress, subnetMaskLen, router)

Changes the configuration of the network interface to use a static IP address.

network→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YNetwork.FindNetwork() YNetwork
yFindNetwork()YNetwork.FindNetwork()

Retrieves a network interface for a given identifier.

```
YNetwork FindNetwork( String func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the network interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YNetwork.isOnline()` to test if the network interface is indeed online at a given time. In case of ambiguity when looking for a network interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the network interface

Returns :

a `YNetwork` object allowing you to drive the network interface.

YNetwork.FirstNetwork() YNetwork yFirstNetwork()YNetwork.FirstNetwork()

Starts the enumeration of network interfaces currently accessible.

YNetwork **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() YNetwork
network.callbackLogin()

Connects to the notification callback and saves the credentials required to log into it.

int **callbackLogin**(String **username**, String **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.

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 network interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

network→get_adminPassword() YNetwork
network→adminPassword()
network.get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

String **get_adminPassword**( )

Returns :

a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns Y_ADMINPASSWORD_INVALID.

network→get_advertisedValue() YNetwork
network→advertisedValue()
network.get_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the network interface (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

network→get_callbackCredentials() YNetwork
network→callbackCredentials()
network.get_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

String **get_callbackCredentials**( )

Returns :
a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns `Y_CALLBACKCREDENTIALS_INVALID`.

network→get_callbackEncoding() YNetwork
network→callbackEncoding()
network.get_callbackEncoding()

Returns the encoding standard to use for representing notification values.

int **get_callbackEncoding**( )

Returns :

a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV and Y_CALLBACKENCODING_YOCTO_API corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns Y_CALLBACKENCODING_INVALID.

network→get_callbackMaxDelay() YNetwork
network→callbackMaxDelay()
network.get_callbackMaxDelay()

Returns the maximum waiting time between two callback notifications, in seconds.

int **get_callbackMaxDelay**( )

Returns :
an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMAXDELAY_INVALID.

network→get_callbackMethod() YNetwork
network→callbackMethod()
network.get_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

int **get_callbackMethod**( )

Returns :

a value among Y_CALLBACKMETHOD_POST, Y_CALLBACKMETHOD_GET and Y_CALLBACKMETHOD_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns Y_CALLBACKMETHOD_INVALID.

network→get_callbackMinDelay() YNetwork
network→callbackMinDelay()
network.get_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

int **get_callbackMinDelay**( )

Returns :
an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMINDELAY_INVALID.

network→get_callbackUrl() YNetwork
network→callbackUrl()
network.get_callbackUrl()

Returns the callback URL to notify of significant state changes.

String **get_callbackUrl**( )

Returns :
a string corresponding to the callback URL to notify of significant state changes

On failure, throws an exception or returns Y_CALLBACKURL_INVALID.

network→get_discoverable() YNetwork
network→discoverable()
network.get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

int **get_discoverable**( )

Returns :

either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns Y_DISCOVERABLE_INVALID.

network→get_errorMessage() YNetwork
network→errorMessage()
network.get_errorMessage()

Returns the error message of the latest error with the network interface.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the network interface object

network→get_errorType() YNetwork
network→errorType()network.get_errorType()

Returns the numerical error code of the latest error with the network interface.

int **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.get_friendlyName()

Returns a global identifier of the network interface in the format MODULE_NAME.FUNCTION_NAME.

String **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() YNetwork
network→functionDescriptor()
network.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

network→get_functionId() YNetwork
network→functionId()network.get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the network interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

network→get_hardwareId() YNetwork
network→hardwareId()network.get_hardwareId()

Returns the unique hardware identifier of the network interface in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the network interface. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the network interface (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

network→get_ipAddress() YNetwork
network→ipAddress()network.get_ipAddress()

Returns the IP address currently in use by the device.

String **get_ipAddress**( )

The address may have been configured statically, or provided by a DHCP server.

Returns :
a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns Y_IPADDRESS_INVALID.

network→get_logicalName() YNetwork
network→logicalName()
network.get_logicalName()

Returns the logical name of the network interface.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the network interface. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

network→get_macAddress() YNetwork
network→macAddress()
network.get_macAddress()

Returns the MAC address of the network interface.

String **get_macAddress**( )

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

Returns :
a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns Y_MACADDRESS_INVALID.

network→get_module() YNetwork
network→module()network.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

network→get_poeCurrent() YNetwork
network→poeCurrent()network.get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

int **get_poeCurrent**( )

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

Returns :
an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns Y_POECURRENT_INVALID.

network→get_primaryDNS() YNetwork
network→primaryDNS()
network.get_primaryDNS()

Returns the IP address of the primary name server to be used by the module.

String **get_primaryDNS**( )

Returns :
a string corresponding to the IP address of the primary name server to be used by the module

On failure, throws an exception or returns Y_PRIMARYDNS_INVALID.

network→get_readiness() YNetwork
network→readiness()network.get_readiness()

Returns the current established working mode of the network interface.

int **get_readiness**( )

Level zero (DOWN_0) means that no hardware link has been detected. Either there is no signal on the network cable, or the selected wireless access point cannot be detected. Level 1 (LIVE_1) is reached when the network is detected, but is not yet connected. For a wireless network, this shows that the requested SSID is present. Level 2 (LINK_2) is reached when the hardware connection is established. For a wired network connection, level 2 means that the cable is attached at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS_4) is reached when the DNS server is reachable on the network. Level 5 (WWW_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

Returns :

a value among `Y_READINESS_DOWN`, `Y_READINESS_EXISTS`, `Y_READINESS_LINKED`, `Y_READINESS_LAN_OK` and `Y_READINESS_WWW_OK` corresponding to the current established working mode of the network interface

On failure, throws an exception or returns `Y_READINESS_INVALID`.

network→get_router() YNetwork
network→router()network.get_router()

Returns the IP address of the router on the device subnet (default gateway).

String **get_router**( )

Returns :
a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns Y_ROUTER_INVALID.

network→get_secondaryDNS() YNetwork
network→secondaryDNS()
network.get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

String **get_secondaryDNS**( )

Returns :
a string corresponding to the IP address of the secondary name server to be used by the module

On failure, throws an exception or returns Y_SECONDARYDNS_INVALID.

network→get_subnetMask() YNetwork
network→subnetMask()
network.get_subnetMask()

Returns the subnet mask currently used by the device.

String **get_subnetMask**( )

Returns :
a string corresponding to the subnet mask currently used by the device

On failure, throws an exception or returns Y_SUBNETMASK_INVALID.

network→get_userData() YNetwork
network→userData()network.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

network→get_userPassword() YNetwork
network→userPassword()
network.get_userPassword()

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

String **get_userPassword**( )

Returns :

a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns Y_USERPASSWORD_INVALID.

network→get_wwwWatchdogDelay() YNetwork
network→wwwWatchdogDelay()
network.get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

int **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.

boolean **isOnline**( )

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

Returns :

`true` if the network interface can be reached, and `false` otherwise

network→load()network.load() YNetwork

Preloads the network interface cache with a specified validity duration.

int **load**(long **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→nextNetwork()`network.nextNetwork()` YNetwork

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

YNetwork **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.

String **ping**(String **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() YNetwork
network.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

network→set_adminPassword() YNetwork
network→setAdminPassword()
network.set_adminPassword()

Changes the password for the "admin" user.

int **set_adminPassword**(String **newval**)

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "admin" user

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackCredentials() YNetwork
network→setCallbackCredentials()
network.set_callbackCredentials()

Changes the credentials required to connect to the callback address.

int **set_callbackCredentials**(String **newval**)

The credentials must be provided as returned by function `get_callbackCredentials`, in the form `username:hash`. The method used to compute the hash varies according to the the authentication scheme implemented by the callback, For Basic authentication, the hash is the MD5 of the string `username:password`. For Digest authentication, the hash is the MD5 of the string `username:realm:password`. For a simpler way to configure callback credentials, use function `callbackLogin` instead. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the credentials required to connect to the callback address

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackEncoding() YNetwork
network→setCallbackEncoding()
network.set_callbackEncoding()

Changes the encoding standard to use for representing notification values.

int **set_callbackEncoding**(int **newval**)

Parameters :

newval a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV and Y_CALLBACKENCODING_YOCTO_API corresponding to the encoding standard to use for representing notification values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMaxDelay() YNetwork
network→setCallbackMaxDelay()
network.set_callbackMaxDelay()

Changes the maximum waiting time between two callback notifications, in seconds.

int **set_callbackMaxDelay**(int **newval**)

Parameters :

newval an integer corresponding to the maximum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMethod() YNetwork
network→setCallbackMethod()
network.set_callbackMethod()

Changes the HTTP method used to notify callbacks for significant state changes.

int **set_callbackMethod**(int **newval**)

Parameters :

newval a value among Y_CALLBACKMETHOD_POST, Y_CALLBACKMETHOD_GET and Y_CALLBACKMETHOD_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMinDelay() YNetwork
network→setCallbackMinDelay()
network.set_callbackMinDelay()

Changes the minimum waiting time between two callback notifications, in seconds.

int **set_callbackMinDelay**(int **newval**)

Parameters :

newval an integer corresponding to the minimum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackUrl() YNetwork
network→setCallbackUrl()
network.set_callbackUrl()

Changes the callback URL to notify significant state changes.

int **set_callbackUrl**(String **newval**)

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the callback URL to notify significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_discoverable() YNetwork
network→setDiscoverable()
network.set_discoverable()

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

int **set_discoverable**(int **newval**)

Parameters :

newval either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_logicalName() YNetwork
network→setLogicalName()
network.set_logicalName()

Changes the logical name of the network interface.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the network interface.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

network→set_primaryDNS() YNetwork
network→setPrimaryDNS()
network.set_primaryDNS()

Changes the IP address of the primary name server to be used by the module.

int **set_primaryDNS**(String **newval**)

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the primary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_secondaryDNS() YNetwork
network→setSecondaryDNS()
network.set_secondaryDNS()

Changes the IP address of the secondary name server to be used by the module.

int **set_secondaryDNS**(String **newval**)

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the secondary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_userData() YNetwork
network→setUserData()network.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

network→set_userPassword() YNetwork
network→setUserPassword()
network.set_userPassword()

Changes the password for the "user" user.

int **set_userPassword**(String **newval**)

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "user" user

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_wwwWatchdogDelay() YNetwork
network→setWwwWatchdogDelay()
network.set_wwwWatchdogDelay()

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

int **set_wwwWatchdogDelay**(int **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.

```
int useDHCP( String fallbackIpAddr,
             int fallbackSubnetMaskLen,
             String 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.

```
int useStaticIP( String ipAddress,
                 int subnetMaskLen,
                 String router)
```

Remember to call the saveToFlash() method and then to reboot the module to apply this setting.

Parameters :

ipAddress	device IP address
subnetMaskLen	subnet mask length, as an integer (eg. 24 means 255.255.255.0)
router	router IP address (default gateway)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.27. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_oscontrol.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YOsControl = yoctolib.YOsControl;
php     require_once('yocto_oscontrol.php');
cpp     #include "yocto_oscontrol.h"
m       #import "yocto_oscontrol.h"
pas     uses yocto_oscontrol;
vb      yocto_oscontrol.vb
cs      yocto_oscontrol.cs
java    import com.yoctopuce.YoctoAPI.YOsControl;
py      from yocto_oscontrol import *
```

Global functions
yFindOsControl(**func**) Retrieves OS control for a given identifier.
yFirstOsControl() Starts the enumeration of OS control currently accessible.
`YOsControl` methods
oscontrol→describe() Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
oscontrol→get_advertisedValue() Returns the current value of the OS control (no more than 6 characters).
oscontrol→get_errorMessage() Returns the error message of the latest error with the OS control.
oscontrol→get_errorType() Returns the numerical error code of the latest error with the OS control.
oscontrol→get_friendlyName() Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.
oscontrol→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
oscontrol→get_functionId() Returns the hardware identifier of the OS control, without reference to the module.
oscontrol→get_hardwareId() Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.
oscontrol→get_logicalName() Returns the logical name of the OS control.
oscontrol→get_module() Gets the `YModule` object for the device on which the function is located.
oscontrol→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

oscontrol→get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

oscontrol→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

oscontrol→isOnline()

Checks if the OS control is currently reachable, without raising any error.

oscontrol→isOnline_async(**callback**, **context**)

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

oscontrol→load(**msValidity**)

Preloads the OS control cache with a specified validity duration.

oscontrol→load_async(**msValidity**, **callback**, **context**)

Preloads the OS control cache with a specified validity duration (asynchronous version).

oscontrol→nextOsControl()

Continues the enumeration of OS control started using `yFirstOsControl()`.

oscontrol→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

oscontrol→set_logicalName(**newval**)

Changes the logical name of the OS control.

oscontrol→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

oscontrol→shutdown(**secBeforeShutDown**)

Schedules an OS shutdown after a given number of seconds.

oscontrol→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YOsControl.FindOsControl() YOsControl
yFindOsControl()YOsControl.FindOsControl()

Retrieves OS control for a given identifier.

YOsControl **FindOsControl**(String **func**)

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() YOsControl yFirstOsControl()YOsControl.FirstOsControl()

Starts the enumeration of OS control currently accessible.

YOsControl **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.

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 OS control (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

oscontrol→get_advertisedValue() YOsControl
oscontrol→advertisedValue()
oscontrol.get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the OS control (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

oscontrol→get_errorMessage() YOsControl
oscontrol→errorMessage()
oscontrol.get_errorMessage()

Returns the error message of the latest error with the OS control.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the OS control object

oscontrol→get_errorType() YOsControl
oscontrol→errorType()
oscontrol.get_errorType()

Returns the numerical error code of the latest error with the OS control.

int **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() YOsControl
oscontrol→friendlyName()
oscontrol.get_friendlyName()

Returns a global identifier of the OS control in the format MODULE_NAME.FUNCTION_NAME.

String **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() YOsControl
oscontrol→functionDescriptor()
oscontrol.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

oscontrol→get_functionId() YOsControl
oscontrol→functionId()
oscontrol.get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the OS control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

oscontrol→get_hardwareId() YOsControl
oscontrol→hardwareId()
oscontrol.get_hardwareId()

Returns the unique hardware identifier of the OS control in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the OS control. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the OS control (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

oscontrol→get_logicalName() YOsControl
oscontrol→logicalName()
oscontrol.get_logicalName()

Returns the logical name of the OS control.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the OS control. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

oscontrol→get_module() YOsControl
oscontrol→module()oscontrol.get_module()

Gets the YModule object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

oscontrol→get_shutdownCountdown() YOsControl
oscontrol→shutdownCountdown()
oscontrol.get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

int **get_shutdownCountdown**( )

Returns :

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns `Y_SHUTDOWNCOUNTDOWN_INVALID`.

oscontrol→get_userData() YOsControl
oscontrol→userData()oscontrol.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

Returns :
`true` if the OS control can be reached, and `false` otherwise

oscontrol→load()oscontrol.load() YOsControl

Preloads the OS control cache with a specified validity duration.

int **load**(long **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→nextOsControl() YOsControl
oscontrol.nextOsControl()

Continues the enumeration of OS control started using `yFirstOsControl()`.

YOsControl **nextOsControl**( )

Returns :

a pointer to a `YOsControl` object, corresponding to OS control currently online, or a `null` pointer if there are no more OS control to enumerate.

oscontrol→registerValueCallback() YOsControl
oscontrol.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

oscontrol→set_logicalName() YOsControl
oscontrol→setLogicalName()
oscontrol.set_logicalName()

Changes the logical name of the OS control.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the OS control.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

oscontrol→set_userData() YOsControl
oscontrol→setUserData()
oscontrol.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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.

int **shutdown**(int **secBeforeShutDown**)

Parameters :

secBeforeShutDown number of seconds before shutdown

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

3.28. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_power.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPower = yoctolib.YPower;
php     require_once('yocto_power.php');
cpp     #include "yocto_power.h"
m       #import "yocto_power.h"
pas     uses yocto_power;
vb      yocto_power.vb
cs      yocto_power.cs
java    import com.yoctopuce.YoctoAPI.YPower;
py      from yocto_power import *
```

Global functions
yFindPower(**func**) Retrieves a electrical power sensor for a given identifier.
yFirstPower() Starts the enumeration of electrical power sensors currently accessible.
YPower methods
power→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
power→describe() Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
power→get_advertisedValue() Returns the current value of the electrical power sensor (no more than 6 characters).
power→get_cosPhi() Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).
power→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
power→get_currentValue() Returns the current measure for the electrical power.
power→get_errorMessage() Returns the error message of the latest error with the electrical power sensor.
power→get_errorType() Returns the numerical error code of the latest error with the electrical power sensor.
power→get_friendlyName() Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.
power→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

power→get_hardwareId()

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

power→get_highestValue()

Returns the maximal value observed for the electrical power.

power→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→get_logicalName()

Returns the logical name of the electrical power sensor.

power→get_lowestValue()

Returns the minimal value observed for the electrical power.

power→get_meter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

power→get_meterTimer()

Returns the elapsed time since last energy counter reset, in seconds.

power→get_module()

Gets the `YModule` object for the device on which the function is located.

power→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

power→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

power→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→get_resolution()

Returns the resolution of the measured values.

power→get_unit()

Returns the measuring unit for the electrical power.

power→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

power→isOnline()

Checks if the electrical power sensor is currently reachable, without raising any error.

power→isOnline_async(**callback**, **context**)

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

power→load(**msValidity**)

Preloads the electrical power sensor cache with a specified validity duration.

power→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

power→load_async(**msValidity**, **callback**, **context**)

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

power→nextPower()

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

power→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

power→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

power→reset()

Resets the energy counter.

power→set_highestValue(**newval**)

Changes the recorded maximal value observed pour the electrical power.

power→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

power→set_logicalName(**newval**)

Changes the logical name of the electrical power sensor.

power→set_lowestValue(**newval**)

Changes the recorded minimal value observed pour the electrical power.

power→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

power→set_resolution(**newval**)

Changes the resolution of the measured values.

power→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

power→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPower.FindPower() YPower yFindPower()YPower.FindPower()

Retrieves a electrical power sensor for a given identifier.

YPower **FindPower**(String **func**)

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() YPower
yFirstPower()YPower.FirstPower()

Starts the enumeration of electrical power sensors currently accessible.

```
YPower 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() YPower
power.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                                                ArrayList<Double> 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.

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 electrical power sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

power→get_advertisedValue() YPower
power→advertisedValue()
power.get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the electrical power sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

power→get_cosPhi() YPower
power→cosPhi()power.get_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

double **get_cosPhi**( )

Returns :

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns Y_COSPHI_INVALID.

power→get_currentRawValue() YPower
power→currentRawValue()
power.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YPower
power→currentValue()power.get_currentValue()

Returns the current measure for the electrical power.

double **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() YPower
power→errorMessage()
power.get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the electrical power sensor object

power→get_errorType() YPower
power→errorType()power.get_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

```
int 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() YPower
power→friendlyName()power.get_friendlyName()

Returns a global identifier of the electrical power sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YPower
power→functionDescriptor()
power.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

power→get_functionId() YPower
power→functionId()power.get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the electrical power sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

power→get_hardwareId() YPower
power→hardwareId()power.get_hardwareId()

Returns the unique hardware identifier of the electrical power sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the electrical power sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the electrical power sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

power→get_highestValue() YPower
power→highestValue()power.get_highestValue()

Returns the maximal value observed for the electrical power.

double **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() YPower
power→logFrequency()
power.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

power→get_logicalName() YPower
power→logicalName()`power.get_logicalName()`

Returns the logical name of the electrical power sensor.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the electrical power sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

power→get_lowestValue() YPower
power→lowestValue()power.get_lowestValue()

Returns the minimal value observed for the electrical power.

double **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() YPower
power→meter()power.get_meter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

double **get_meter**( )

Note that this counter is reset at each start of the device.

Returns :
a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns Y_METER_INVALID.

power→get_meterTimer() YPower
power→meterTimer()power.get_meterTimer()

Returns the elapsed time since last energy counter reset, in seconds.

int **get_meterTimer**( )

Returns :
an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns Y_METERTIMER_INVALID.

power→get_module() YPower
power→module()power.get_module()

Gets the YModule object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

power→get_recordedData() YPower
power→recordedData()
power.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

power→get_reportFrequency() YPower
power→reportFrequency()
power.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

power→get_resolution() YPower
power→resolution()power.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

power→get_unit() YPower
power→unit()power.get_unit()

Returns the measuring unit for the electrical power.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns Y_UNIT_INVALID.

power→get_userData() YPower
power→userData()power.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

Returns :
`true` if the electrical power sensor can be reached, and `false` otherwise

power→load()power.load() YPower

Preloads the electrical power sensor cache with a specified validity duration.

int **load**(long **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() YPower
power.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

int **loadCalibrationPoints**(ArrayList<Double> **rawValues**,
ArrayList<Double> **refValues**)

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→nextPower()power.nextPower() YPower

Continues the enumeration of electrical power sensors started using yFirstPower().

YPower **nextPower**( )

Returns :

a pointer to a YPower object, corresponding to a electrical power sensor currently online, or a null pointer if there are no more electrical power sensors to enumerate.

power→registerTimedReportCallback() YPower
power.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

power→registerValueCallback() YPower
power.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback 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.

int **reset**( )

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_highestValue() YPower
power→setHighestValue()
power.set_highestValue()

Changes the recorded maximal value observed pour the electrical power.

int **set_highestValue**(double **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() YPower
power→setLogFrequency()
power.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logicalName() YPower
power→setLogicalName()
power.set_logicalName()

Changes the logical name of the electrical power sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the electrical power sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

power→set_lowestValue() YPower
power→setLowestValue()
power.set_lowestValue()

Changes the recorded minimal value observed pour the electrical power.

int **set_lowestValue**(double **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() YPower
power→setReportFrequency()
power.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_resolution() YPower
power→setResolution()power.set_resolution()

Changes the resolution of the measured values.

int **set_resolution**(double **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() YPower
power→setUserData()power.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.29. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pressure.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPressure = yoctolib.YPressure;
php     require_once('yocto_pressure.php');
cpp     #include "yocto_pressure.h"
m       #import "yocto_pressure.h"
pas     uses yocto_pressure;
vb      yocto_pressure.vb
cs      yocto_pressure.cs
java    import com.yoctopuce.YoctoAPI.YPressure;
py      from yocto_pressure import *
```

Global functions
yFindPressure(**func**) Retrieves a pressure sensor for a given identifier.
yFirstPressure() Starts the enumeration of pressure sensors currently accessible.
YPressure methods
pressure→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
pressure→describe() Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pressure→get_advertisedValue() Returns the current value of the pressure sensor (no more than 6 characters).
pressure→get_currentRawValue() Returns the unrounded and uncalibrated raw value returned by the sensor.
pressure→get_currentValue() Returns the current measure for the pressure.
pressure→get_errorMessage() Returns the error message of the latest error with the pressure sensor.
pressure→get_errorType() Returns the numerical error code of the latest error with the pressure sensor.
pressure→get_friendlyName() Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.
pressure→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
pressure→get_functionId() Returns the hardware identifier of the pressure sensor, without reference to the module.
pressure→get_hardwareId() Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

pressure→get_highestValue()

Returns the maximal value observed for the pressure.

pressure→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pressure→get_logicalName()

Returns the logical name of the pressure sensor.

pressure→get_lowestValue()

Returns the minimal value observed for the pressure.

pressure→get_module()

Gets the `YModule` object for the device on which the function is located.

pressure→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pressure→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pressure→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pressure→get_resolution()

Returns the resolution of the measured values.

pressure→get_unit()

Returns the measuring unit for the pressure.

pressure→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pressure→isOnline()

Checks if the pressure sensor is currently reachable, without raising any error.

pressure→isOnline_async(**callback**, **context**)

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

pressure→load(**msValidity**)

Preloads the pressure sensor cache with a specified validity duration.

pressure→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

pressure→load_async(**msValidity**, **callback**, **context**)

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

pressure→nextPressure()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

pressure→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

pressure→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

pressure→set_highestValue(**newval**)

Changes the recorded maximal value observed for the pressure.

pressure→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

pressure→set_logicalName(**newval**)

Changes the logical name of the pressure sensor.

pressure→set_lowestValue(newval)

Changes the recorded minimal value observed for the pressure.

pressure→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

pressure→set_resolution(newval)

Changes the resolution of the measured physical values.

pressure→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pressure→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPressure.FindPressure() YPressure yFindPressure()YPressure.FindPressure()

Retrieves a pressure sensor for a given identifier.

YPressure **FindPressure**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the pressure sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPressure.isOnline()` to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the pressure sensor

Returns :

a `YPressure` object allowing you to drive the pressure sensor.

YPressure.FirstPressure() YPressure yFirstPressure()YPressure.FirstPressure()

Starts the enumeration of pressure sensors currently accessible.

YPressure **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() YPressure
pressure.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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 pressure sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

pressure→get_advertisedValue() YPressure
pressure→advertisedValue()
pressure.get_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the pressure sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pressure→get_currentRawValue() YPressure
pressure→currentRawValue()
pressure.get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **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() YPressure
pressure→currentValue()
pressure.get_currentValue()

Returns the current measure for the pressure.

double **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() YPressure
pressure→errorMessage()
pressure.get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the pressure sensor object

pressure→get_errorType() YPressure
pressure→errorType()`pressure.get_errorType()`

Returns the numerical error code of the latest error with the pressure sensor.

```
int 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() YPressure
pressure→friendlyName()
pressure.get_friendlyName()

Returns a global identifier of the pressure sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YPressure
pressure→functionDescriptor()
pressure.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pressure→get_functionId() YPressure
pressure→functionId()
pressure.get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the pressure sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pressure→get_hardwareId() YPressure
pressure→hardwareId()
pressure.get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the pressure sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the pressure sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pressure→get_highestValue() YPressure
pressure→highestValue()
pressure.get_highestValue()

Returns the maximal value observed for the pressure.

double **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() YPressure
pressure→logFrequency()
pressure.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

pressure→get_logicalName() YPressure
pressure→logicalName()
pressure.get_logicalName()

Returns the logical name of the pressure sensor.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the pressure sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pressure→get_lowestValue() YPressure
pressure→lowestValue()
pressure.get_lowestValue()

Returns the minimal value observed for the pressure.

double **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() YPressure
pressure→module()pressure.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

pressure→get_recordedData() YPressure
pressure→recordedData()
pressure.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pressure→get_reportFrequency() YPressure
pressure→reportFrequency()
pressure.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

pressure→get_resolution() YPressure
pressure→resolution()
pressure.get_resolution()

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

pressure→get_unit() YPressure
pressure→unit()`pressure.get_unit()`

Returns the measuring unit for the pressure.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

pressure→get_userData() YPressure
pressure→userData()`pressure.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

Returns :
`true` if the pressure sensor can be reached, and `false` otherwise

pressure→load()pressure.load() YPressure

Preloads the pressure sensor cache with a specified validity duration.

int **load**(long **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() YPressure
pressure.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→nextPressure() YPressure
pressure.nextPressure()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

YPressure **nextPressure**( )

Returns :

a pointer to a `YPressure` object, corresponding to a pressure sensor currently online, or a `null` pointer if there are no more pressure sensors to enumerate.

pressure→registerTimedReportCallback() YPressure
pressure.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

pressure→registerValueCallback() YPressure
pressure.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pressure→set_highestValue() YPressure
pressure→setHighestValue()
pressure.set_highestValue()

Changes the recorded maximal value observed for the pressure.

int **set_highestValue**(double **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() YPressure
pressure→setLogFrequency()
pressure.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logicalName() YPressure
pressure→setLogicalName()
pressure.set_logicalName()

Changes the logical name of the pressure sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the pressure sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

pressure→set_lowestValue() YPressure
pressure→setLowestValue()
pressure.set_lowestValue()

Changes the recorded minimal value observed for the pressure.

int **set_lowestValue**(double **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() YPressure
pressure→setReportFrequency()
pressure.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_resolution() YPressure
pressure→setResolution()
pressure.set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_userData() YPressure
pressure→setUserData()
pressure.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.30. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pwmoutput.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPwmOutput = yoctolib.YPwmOutput;
php     require_once('yocto_pwmoutput.php');
cpp     #include "yocto_pwmoutput.h"
m       #import "yocto_pwmoutput.h"
pas     uses yocto_pwmoutput;
vb      yocto_pwmoutput.vb
cs      yocto_pwmoutput.cs
java    import com.yoctopuce.YoctoAPI.YPwmOutput;
py      from yocto_pwmoutput import *
```

Global functions
yFindPwmOutput(**func**) Retrieves a PWM for a given identifier.
yFirstPwmOutput() Starts the enumeration of PWMs currently accessible.
`YPwmOutput` methods
pwmoutput→describe() Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pwmoutput→dutyCycleMove(**target**, **ms_duration**) Performs a smooth change of the pulse duration toward a given value.
pwmoutput→get_advertisedValue() Returns the current value of the PWM (no more than 6 characters).
pwmoutput→get_dutyCycle() Returns the PWM duty cycle, in per cents.
pwmoutput→get_dutyCycleAtPowerOn() Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100
pwmoutput→get_enabled() Returns the state of the PWMs.
pwmoutput→get_enabledAtPowerOn() Returns the state of the PWM at device power on.
pwmoutput→get_errorMessage() Returns the error message of the latest error with the PWM.
pwmoutput→get_errorType() Returns the numerical error code of the latest error with the PWM.
pwmoutput→get_frequency() Returns the PWM frequency in Hz.
pwmoutput→get_friendlyName() Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.
pwmoutput→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

pwmoutput→get_functionId()
Returns the hardware identifier of the PWM, without reference to the module.

pwmoutput→get_hardwareId()
Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

pwmoutput→get_logicalName()
Returns the logical name of the PWM.

pwmoutput→get_module()
Gets the YModule object for the device on which the function is located.

pwmoutput→get_module_async(**callback**, **context**)
Gets the YModule object for the device on which the function is located (asynchronous version).

pwmoutput→get_period()
Returns the PWM period in milliseconds.

pwmoutput→get_pulseDuration()
Returns the PWM pulse length in milliseconds.

pwmoutput→get_userData()
Returns the value of the userData attribute, as previously stored using method set_userData.

pwmoutput→isOnline()
Checks if the PWM is currently reachable, without raising any error.

pwmoutput→isOnline_async(**callback**, **context**)
Checks if the PWM is currently reachable, without raising any error (asynchronous version).

pwmoutput→load(**msValidity**)
Preloads the PWM cache with a specified validity duration.

pwmoutput→load_async(**msValidity**, **callback**, **context**)
Preloads the PWM cache with a specified validity duration (asynchronous version).

pwmoutput→nextPwmOutput()
Continues the enumeration of PWMs started using yFirstPwmOutput().

pwmoutput→pulseDurationMove(**ms_target**, **ms_duration**)
Performs a smooth transistion of the pulse duration toward a given value.

pwmoutput→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

pwmoutput→set_dutyCycle(**newval**)
Changes the PWM duty cycle, in per cents.

pwmoutput→set_dutyCycleAtPowerOn(**newval**)
Changes the PWM duty cycle at device power on.

pwmoutput→set_enabled(**newval**)
Stops or starts the PWM.

pwmoutput→set_enabledAtPowerOn(**newval**)
Changes the state of the PWM at device power on.

pwmoutput→set_frequency(**newval**)
Changes the PWM frequency.

pwmoutput→set_logicalName(**newval**)
Changes the logical name of the PWM.

pwmoutput→set_period(**newval**)
Changes the PWM period.

pwmoutput→set_pulseDuration(newval)

Changes the PWM pulse length, in milliseconds.

pwmoutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmoutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmOutput.FindPwmOutput() YPwmOutput yFindPwmOutput()YPwmOutput.FindPwmOutput()

Retrieves a PWM for a given identifier.

YPwmOutput **FindPwmOutput**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the PWM

Returns :

a `YPwmOutput` object allowing you to drive the PWM.

YPwmOutput.FirstPwmOutput() YPwmOutput yFirstPwmOutput()YPwmOutput.FirstPwmOutput()

Starts the enumeration of PWMs currently accessible.

```
YPwmOutput 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.

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 PWM (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

pwmoutput→dutyCycleMove() YPwmOutput
pwmoutput.dutyCycleMove()

Performs a smooth change of the pulse duration toward a given value.

int **dutyCycleMove**(double **target**, int **ms_duration**)

Parameters :

target	new duty cycle at the end of the transition (floating-point number, between 0 and 1)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→get_advertisedValue() YPwmOutput
pwmoutput→advertisedValue()
pwmoutput.get_advertisedValue()

Returns the current value of the PWM (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the PWM (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwmoutput→get_dutyCycle() YPwmOutput
pwmoutput→dutyCycle()
pwmoutput.get_dutyCycle()

Returns the PWM duty cycle, in per cents.

double **get_dutyCycle**( )

Returns :
a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns Y_DUTYCYCLE_INVALID.

pwmoutput→get_dutyCycleAtPowerOn() YPwmOutput
pwmoutput→dutyCycleAtPowerOn()
pwmoutput.get_dutyCycleAtPowerOn()

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

double **get_dutyCycleAtPowerOn**( )

Returns :
a floating point number corresponding to the PWMs duty cycle at device power on as a floating point number between 0 and 100

On failure, throws an exception or returns Y_DUTYCYCLEATPOWERON_INVALID.

pwmoutput→get_enabled() YPwmOutput
pwmoutput→enabled()pwmoutput.get_enabled()

Returns the state of the PWMs.

```
int get_enabled( )
```

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the PWMs

On failure, throws an exception or returns Y_ENABLED_INVALID.

pwmoutput→get_enabledAtPowerOn() YPwmOutput
pwmoutput→enabledAtPowerOn()
pwmoutput.get_enabledAtPowerOn()

Returns the state of the PWM at device power on.

int **get_enabledAtPowerOn**( )

Returns :

either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the state of the PWM at device power on

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.

pwmoutput→get_errorMessage() YPwmOutput
pwmoutput→errorMessage()
pwmoutput.get_errorMessage()

Returns the error message of the latest error with the PWM.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the PWM object

pwmoutput→get_errorType() YPwmOutput
pwmoutput→errorType()
pwmoutput.get_errorType()

Returns the numerical error code of the latest error with the PWM.

```
int 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() YPwmOutput
pwmoutput→frequency()
pwmoutput.get_frequency()

Returns the PWM frequency in Hz.

int **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() YPwmOutput
pwmoutput→friendlyName()
pwmoutput.get_friendlyName()

Returns a global identifier of the PWM in the format MODULE_NAME.FUNCTION_NAME.

String **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() YPwmOutput
pwmoutput→functionDescriptor()
pwmoutput.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmoutput→get_functionId() YPwmOutput
pwmoutput→functionId()
pwmoutput.get_functionId()

Returns the hardware identifier of the PWM, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the PWM (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwmoutput→get_hardwareId() YPwmOutput
pwmoutput→hardwareId()
pwmoutput.get_hardwareId()

Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the PWM (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pwmoutput→get_logicalName() YPwmOutput
pwmoutput→logicalName()
pwmoutput.get_logicalName()

Returns the logical name of the PWM.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the PWM. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pwmoutput→get_module() YPwmOutput
pwmoutput→module()pwmoutput.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

pwmoutput→get_period() YPwmOutput
pwmoutput→period()`pwmoutput.get_period()`

Returns the PWM period in milliseconds.

double **get_period**( )

Returns :
a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

pwmoutput→get_pulseDuration() YPwmOutput
pwmoutput→pulseDuration()
pwmoutput.get_pulseDuration()

Returns the PWM pulse length in milliseconds.

double **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() YPwmOutput
pwmoutput→userData()
pwmoutput.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

Returns :

`true` if the PWM can be reached, and `false` otherwise

pwmoutput→load()pwmoutput.load() YPwmOutput

Preloads the PWM cache with a specified validity duration.

int **load**(long **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→nextPwmOutput() YPwmOutput
pwmoutput.nextPwmOutput()

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

YPwmOutput **nextPwmOutput**( )

Returns :

a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.

pwmoutput→pulseDurationMove() YPwmOutput
pwmoutput.pulseDurationMove()

Performs a smooth transistion of the pulse duration toward a given value.

int **pulseDurationMove**(double **ms_target**, int **ms_duration**)

Any period, frequency, duty cycle or pulse width change will cancel any ongoing transition process.

Parameters :

ms_target	new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→registerValueCallback() YPwmOutput
pwmoutput.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmoutput→set_dutyCycle() YPwmOutput
pwmoutput→setDutyCycle()
pwmoutput.set_dutyCycle()

Changes the PWM duty cycle, in per cents.

int **set_dutyCycle**(double **newval**)

Parameters :

newval a floating point number corresponding to the PWM duty cycle, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_dutyCycleAtPowerOn() YPwmOutput
pwmoutput→setDutyCycleAtPowerOn()
pwmoutput.set_dutyCycleAtPowerOn()

Changes the PWM duty cycle at device power on.

int **set_dutyCycleAtPowerOn**(double **newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a floating point number corresponding to the PWM duty cycle at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabled() YPwmOutput
pwmoutput→setEnabled()
pwmoutput.set_enabled()

Stops or starts the PWM.

int **set_enabled**(int **newval**)

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabledAtPowerOn() YPwmOutput
pwmoutput→setEnabledAtPowerOn()
pwmoutput.set_enabledAtPowerOn()

Changes the state of the PWM at device power on.

int **set_enabledAtPowerOn**(int **newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_frequency() YPwmOutput
pwmoutput→setFrequency()
pwmoutput.set_frequency()

Changes the PWM frequency.

int **set_frequency**(int **newval**)

The duty cycle is kept unchanged thanks to an automatic pulse width change.

Parameters :

newval an integer corresponding to the PWM frequency

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_logicalName() YPwmOutput
pwmoutput→setLogicalName()
pwmoutput.set_logicalName()

Changes the logical name of the PWM.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the PWM.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→set_period() YPwmOutput
pwmoutput→setPeriod()pwmoutput.set_period()

Changes the PWM period.

int **set_period**(double **newval**)

Parameters :

newval a floating point number corresponding to the PWM period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_pulseDuration() YPwmOutput
pwmoutput→setPulseDuration()
pwmoutput.set_pulseDuration()

Changes the PWM pulse length, in milliseconds.

int **set_pulseDuration**(double **newval**)

A pulse length cannot be longer than period, otherwise it is truncated.

Parameters :

newval a floating point number corresponding to the PWM pulse length, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_userData() YPwmOutput
pwmoutput→setUserData()
pwmoutput.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.31. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pwmpowersource.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPwmPowerSource = yoctolib.YPwmPowerSource;
php     require_once('yocto_pwmpowersource.php');
cpp     #include "yocto_pwmpowersource.h"
m       #import "yocto_pwmpowersource.h"
pas     uses yocto_pwmpowersource;
vb      yocto_pwmpowersource.vb
cs      yocto_pwmpowersource.cs
java    import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py      from yocto_pwmpowersource import *
```

Global functions
yFindPwmPowerSource(**func**) Retrieves a voltage source for a given identifier.
yFirstPwmPowerSource() Starts the enumeration of Voltage sources currently accessible.
`YPwmPowerSource` methods
pwmpowersource→describe() Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pwmpowersource→get_advertisedValue() Returns the current value of the voltage source (no more than 6 characters).
pwmpowersource→get_errorMessage() Returns the error message of the latest error with the voltage source.
pwmpowersource→get_errorType() Returns the numerical error code of the latest error with the voltage source.
pwmpowersource→get_friendlyName() Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.
pwmpowersource→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
pwmpowersource→get_functionId() Returns the hardware identifier of the voltage source, without reference to the module.
pwmpowersource→get_hardwareId() Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.
pwmpowersource→get_logicalName() Returns the logical name of the voltage source.
pwmpowersource→get_module() Gets the `YModule` object for the device on which the function is located.
pwmpowersource→get_module_async(**callback**, **context**) Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwmpowersource→get_powerMode()

Returns the selected power source for the PWM on the same device

pwmpowersource→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwmpowersource→isOnline()

Checks if the voltage source is currently reachable, without raising any error.

pwmpowersource→isOnline_async(**callback**, **context**)

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

pwmpowersource→load(**msValidity**)

Preloads the voltage source cache with a specified validity duration.

pwmpowersource→load_async(**msValidity**, **callback**, **context**)

Preloads the voltage source cache with a specified validity duration (asynchronous version).

pwmpowersource→nextPwmPowerSource()

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

pwmpowersource→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

pwmpowersource→set_logicalName(**newval**)

Changes the logical name of the voltage source.

pwmpowersource→set_powerMode(**newval**)

Changes the PWM power source.

pwmpowersource→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

pwmpowersource→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmPowerSource.FindPwmPowerSource() YPwmPowerSource
yFindPwmPowerSource()
YPwmPowerSource.FindPwmPowerSource()

Retrieves a voltage source for a given identifier.

YPwmPowerSource **FindPwmPowerSource**(String **func**)

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() YPwmPowerSource
yFirstPwmPowerSource()
YPwmPowerSource.FirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

```
YPwmPowerSource 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() YPwmPowerSource
pwmpowersource.describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 voltage source (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

pwmpowersource→get_advertisedValue() YPwmPowerSource
pwmpowersource→advertisedValue()
pwmpowersource.get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the voltage source (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwmpowersource→get_errorMessage() YPwmPowerSource
pwmpowersource→errorMessage()
pwmpowersource.get_errorMessage()

Returns the error message of the latest error with the voltage source.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the voltage source object

pwmpowersource→get_errorType() YPwmPowerSource
pwmpowersource→errorType()
pwmpowersource.get_errorType()

Returns the numerical error code of the latest error with the voltage source.

int **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() YPwmPowerSource
pwmpowersource→friendlyName()
pwmpowersource.get_friendlyName()

Returns a global identifier of the voltage source in the format MODULE_NAME.FUNCTION_NAME.

String **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() YPwmPowerSource
pwmpowersource→functionDescriptor()
pwmpowersource.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmpowersource→get_functionId() YPwmPowerSource
pwmpowersource→functionId()
pwmpowersource.get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the voltage source (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwmpowersource→get_hardwareId() YPwmPowerSource
pwmpowersource→hardwareId()
pwmpowersource.get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage source. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the voltage source (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

pwmpowersource→get_logicalName() YPwmPowerSource
pwmpowersource→logicalName()
pwmpowersource.get_logicalName()

Returns the logical name of the voltage source.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the voltage source. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pwmpowersource→get_module() YPwmPowerSource
pwmpowersource→module()
pwmpowersource.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

pwmpowersource→get_powerMode() YPwmPowerSource
pwmpowersource→powerMode()
pwmpowersource.get_powerMode()

Returns the selected power source for the PWM on the same device

int **get_powerMode**( )

Returns :

a value among Y_POWERMODE_USB_5V, Y_POWERMODE_USB_3V, Y_POWERMODE_EXT_V and Y_POWERMODE_OPNDRN corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns Y_POWERMODE_INVALID.

pwmpowersource→get_userData() YPwmPowerSource
pwmpowersource→userData()
pwmpowersource.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

pwmpowersource→isOnline() YPwmPowerSource
pwmpowersource.isOnline()

Checks if the voltage source is currently reachable, without raising any error.

boolean **isOnline**( )

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

Returns :

`true` if the voltage source can be reached, and `false` otherwise

pwmpowersource→load()pwmpowersource.load() YPwmPowerSource

Preloads the voltage source cache with a specified validity duration.

int **load**(long **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→nextPwmPowerSource() YPwmPowerSource
pwmpowersource.nextPwmPowerSource()

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

YPwmPowerSource **nextPwmPowerSource**( )

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more Voltage sources to enumerate.

pwmpowersource→registerValueCallback() YPwmPowerSource
pwmpowersource.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmpowersource→set_logicalName() YPwmPowerSource
pwmpowersource→setLogicalName()
pwmpowersource.set_logicalName()

Changes the logical name of the voltage source.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage source.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

pwmpowersource→set_powerMode() YPwmPowerSource
pwmpowersource→setPowerMode()
pwmpowersource.set_powerMode()

Changes the PWM power source.

int **set_powerMode**(int **newval**)

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

Parameters :

newval a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the PWM power source

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→set_userData() YPwmPowerSource
pwmpowersource→setUserData()
pwmpowersource.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.32. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_gyro.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YGyro = yoctolib.YGyro;
php     require_once('yocto_gyro.php');
cpp     #include "yocto_gyro.h"
m       #import "yocto_gyro.h"
pas     uses yocto_gyro;
vb      yocto_gyro.vb
cs      yocto_gyro.cs
java    import com.yoctopuce.YoctoAPI.YGyro;
py      from yocto_gyro import *
```

Global functions
yFindQt(**func**) Retrieves a quaternion component for a given identifier.
yFirstQt() Starts the enumeration of quaternion components currently accessible.
YQt methods
qt→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
qt→describe() Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
qt→get_advertisedValue() Returns the current value of the quaternion component (no more than 6 characters).
qt→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
qt→get_currentValue() Returns the current value of the value.
qt→get_errorMessage() Returns the error message of the latest error with the quaternion component.
qt→get_errorType() Returns the numerical error code of the latest error with the quaternion component.
qt→get_friendlyName() Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.
qt→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
qt→get_functionId() Returns the hardware identifier of the quaternion component, without reference to the module.
qt→get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

qt→get_highestValue()

Returns the maximal value observed for the value since the device was started.

qt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

qt→get_logicalName()

Returns the logical name of the quaternion component.

qt→get_lowestValue()

Returns the minimal value observed for the value since the device was started.

qt→get_module()

Gets the `YModule` object for the device on which the function is located.

qt→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

qt→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

qt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

qt→get_resolution()

Returns the resolution of the measured values.

qt→get_unit()

Returns the measuring unit for the value.

qt→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

qt→isOnline()

Checks if the quaternion component is currently reachable, without raising any error.

qt→isOnline_async(**callback**, **context**)

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

qt→load(**msValidity**)

Preloads the quaternion component cache with a specified validity duration.

qt→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

qt→load_async(**msValidity**, **callback**, **context**)

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

qt→nextQt()

Continues the enumeration of quaternion components started using `yFirstQt()`.

qt→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

qt→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

qt→set_highestValue(**newval**)

Changes the recorded maximal value observed.

qt→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

qt→set_logicalName(**newval**)

Changes the logical name of the quaternion component.

qt→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

qt→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

qt→set_resolution(**newval**)

Changes the resolution of the measured physical values.

qt→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

qt→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YQt.FindQt() YQt
yFindQt()YQt.FindQt()

Retrieves a quaternion component for a given identifier.

```
YQt FindQt( String func)
```

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() YQt yFirstQt()YQt.FirstQt()

Starts the enumeration of quaternion components currently accessible.

```
YQt 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() YQt
qt.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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 quaternion component (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

qt→get_advertisedValue() YQt
qt→advertisedValue()qt.get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the quaternion component (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

qt→get_currentRawValue() YQt
qt→currentRawValue()qt.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YQt
qt→currentValue()qt.get_currentValue()

Returns the current value of the value.

double **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() YQt
qt→errorMessage()qt.get_errorMessage()

Returns the error message of the latest error with the quaternion component.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the quaternion component object

qt→get_errorType() YQt
qt→errorType()qt.get_errorType()

Returns the numerical error code of the latest error with the quaternion component.

```
int 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.get_friendlyName()

Returns a global identifier of the quaternion component in the format MODULE_NAME.FUNCTION_NAME.

String **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() YQt
qt→functionDescriptor()
qt.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

qt→get_functionId() YQt
qt→functionId()`qt.get_functionId()`

Returns the hardware identifier of the quaternion component, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the quaternion component (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

qt→get_hardwareId() YQt
qt→hardwareId()qt.get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quaternion component. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the quaternion component (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

qt→get_highestValue() YQt
qt→highestValue()qt.get_highestValue()

Returns the maximal value observed for the value since the device was started.

double **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

qt→get_logFrequency() YQt
qt→logFrequency()qt.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

qt→get_logicalName() YQt
qt→logicalName()`qt.get_logicalName()`

Returns the logical name of the quaternion component.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the quaternion component. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

qt→get_lowestValue() YQt
qt→lowestValue()qt.get_lowestValue()

Returns the minimal value observed for the value since the device was started.

double **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the value since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

qt→get_module() YQt
qt→module()qt.get_module()

Gets the YModule object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
an instance of YModule

qt→get_recordedData() YQt
qt→recordedData()qt.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

qt→get_reportFrequency() YQt
qt→reportFrequency()qt.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

qt→get_resolution() YQt
qt→resolution()qt.get_resolution()

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

qt→get_unit() YQt
qt→unit()qt.get_unit()

Returns the measuring unit for the value.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the value

On failure, throws an exception or returns Y_UNIT_INVALID.

qt→get_userData() YQt
qt→userData()qt.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

Returns :
`true` if the quaternion component can be reached, and `false` otherwise

qt→load()qt.load() YQt

Preloads the quaternion component cache with a specified validity duration.

int **load**(long **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() YQt
qt.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→nextQt()`qt.nextQt()` YQt

Continues the enumeration of quaternion components started using `yFirstQt()`.

YQt **nextQt**( )

Returns :

a pointer to a `YQt` object, corresponding to a quaternion component currently online, or a `null` pointer if there are no more quaternion components to enumerate.

qt→registerTimedReportCallback() YQt
qt.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

qt→registerValueCallback() YQt
qt.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

qt→set_highestValue() YQt
qt→setHighestValue()qt.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logFrequency() YQt
qt→setLogFrequency()qt.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logicalName() YQt
qt→setLogicalName()qt.set_logicalName()

Changes the logical name of the quaternion component.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the quaternion component.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

qt→set_lowestValue() YQt
qt→setLowestValue()qt.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_reportFrequency() YQt
qt→setReportFrequency()
qt.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_resolution() YQt
qt→setResolution()qt.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_userData() YQt
qt→setUserData()qt.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.33. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_realtimeclock.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YRealTimeClock = yoctolib.YRealTimeClock;
php     require_once('yocto_realtimeclock.php');
cpp     #include "yocto_realtimeclock.h"
m       #import "yocto_realtimeclock.h"
pas     uses yocto_realtimeclock;
vb      yocto_realtimeclock.vb
cs      yocto_realtimeclock.cs
java    import com.yoctopuce.YoctoAPI.YRealTimeClock;
py      from yocto_realtimeclock import *
```

Global functions
yFindRealTimeClock(**func**) Retrieves a clock for a given identifier.
yFirstRealTimeClock() Starts the enumeration of clocks currently accessible.
`YRealTimeClock` methods
realtimeclock→describe() Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
realtimeclock→get_advertisedValue() Returns the current value of the clock (no more than 6 characters).
realtimeclock→get_dateTime() Returns the current time in the form "YYYY/MM/DD hh:mm:ss"
realtimeclock→get_errorMessage() Returns the error message of the latest error with the clock.
realtimeclock→get_errorType() Returns the numerical error code of the latest error with the clock.
realtimeclock→get_friendlyName() Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.
realtimeclock→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
realtimeclock→get_functionId() Returns the hardware identifier of the clock, without reference to the module.
realtimeclock→get_hardwareId() Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.
realtimeclock→get_logicalName() Returns the logical name of the clock.
realtimeclock→get_module()

Gets the YModule object for the device on which the function is located.

realtimeclock→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

realtimeclock→get_timeSet()

Returns true if the clock has been set, and false otherwise.

realtimeclock→get_unixTime()

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

realtimeclock→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

realtimeclock→get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

realtimeclock→isOnline()

Checks if the clock is currently reachable, without raising any error.

realtimeclock→isOnline_async(callback, context)

Checks if the clock is currently reachable, without raising any error (asynchronous version).

realtimeclock→load(msValidity)

Preloads the clock cache with a specified validity duration.

realtimeclock→load_async(msValidity, callback, context)

Preloads the clock cache with a specified validity duration (asynchronous version).

realtimeclock→nextRealTimeClock()

Continues the enumeration of clocks started using yFirstRealTimeClock().

realtimeclock→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

realtimeclock→set_logicalName(newval)

Changes the logical name of the clock.

realtimeclock→set_unixTime(newval)

Changes the current time.

realtimeclock→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

realtimeclock→set_utcOffset(newval)

Changes the number of seconds between current time and UTC time (time zone).

realtimeclock→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRealTimeClock.FindRealTimeClock() YRealTimeClock
yFindRealTimeClock()
YRealTimeClock.FindRealTimeClock()

Retrieves a clock for a given identifier.

YRealTimeClock **FindRealTimeClock**(String **func**)

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() YRealTimeClock
yFirstRealTimeClock()
YRealTimeClock.FirstRealTimeClock()

Starts the enumeration of clocks currently accessible.

YRealTimeClock **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() YRealTimeClock
realtimeclock.describe()

Returns a short text that describes unambiguously the instance of the clock in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 clock (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

realtimeclock→get_advertisedValue() YRealTimeClock
realtimeclock→advertisedValue()
realtimeclock.get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the clock (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

realtimeclock→get_dateTime() YRealTimeClock
realtimeclock→dateTime()
realtimeclock.get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

String **get_dateTime**( )

Returns :
a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns `Y_DATETIME_INVALID`.

realtimeclock→get_errorMessage() YRealTimeClock
realtimeclock→errorMessage()
realtimeclock.get_errorMessage()

Returns the error message of the latest error with the clock.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the clock object

realtimeclock→get_errorType() YRealTimeClock
realtimeclock→errorType()
realtimeclock.get_errorType()

Returns the numerical error code of the latest error with the clock.

int **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() YRealTimeClock
realtimeclock→friendlyName()
realtimeclock.get_friendlyName()

Returns a global identifier of the clock in the format MODULE_NAME.FUNCTION_NAME.

String **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() YRealTimeClock
realtimeclock→functionDescriptor()
realtimeclock.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

realtimeclock→get_functionId() YRealTimeClock
realtimeclock→functionId()
realtimeclock.get_functionId()

Returns the hardware identifier of the clock, without reference to the module.

String **get_functionId**( )

For example relay1

Returns :

a string that identifies the clock (ex: relay1) On failure, throws an exception or returns Y_FUNCTIONID_INVALID.

realtimeclock→get_hardwareId() YRealTimeClock
realtimeclock→hardwareId()
realtimeclock.get_hardwareId()

Returns the unique hardware identifier of the clock in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the clock. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the clock (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

realtimeclock→get_logicalName() YRealTimeClock
realtimeclock→logicalName()
realtimeclock.get_logicalName()

Returns the logical name of the clock.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the clock. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

realtimeclock→get_module() YRealTimeClock
realtimeclock→module()
realtimeclock.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

realtimeclock→get_timeSet() YRealTimeClock
realtimeclock→timeSet()
realtimeclock.get_timeSet()

Returns true if the clock has been set, and false otherwise.

int **get_timeSet**( )

Returns :

either Y_TIMESET_FALSE or Y_TIMESET_TRUE, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns Y_TIMESET_INVALID.

realtimeclock→get_unixTime() YRealTimeClock
realtimeclock→unixTime()
realtimeclock.get_unixTime()

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

long **get_unixTime**( )

Returns :
an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns `Y_UNIXTIME_INVALID`.

realtimeclock→get_userData() YRealTimeClock
realtimeclock→userData()
realtimeclock.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

realtimeclock→get_utcOffset() YRealTimeClock
realtimeclock→utcOffset()
realtimeclock.get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

int **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() YRealTimeClock
`realtimeclock.isOnline()`

Checks if the clock is currently reachable, without raising any error.

boolean **isOnline**( )

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

Returns :
`true` if the clock can be reached, and `false` otherwise

realtimeclock→load()`realtimeclock.load()` YRealTimeClock

Preloads the clock cache with a specified validity duration.

int **load**(long **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→nextRealTimeClock() YRealTimeClock
`realtimeclock.nextRealTimeClock()`

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

YRealTimeClock **nextRealTimeClock**( )

Returns :

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a `null` pointer if there are no more clocks to enumerate.

realtimeclock→registerValueCallback() YRealTimeClock
realtimeclock.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

realtimeclock→set_logicalName() YRealTimeClock
realtimeclock→setLogicalName()
realtimeclock.set_logicalName()

Changes the logical name of the clock.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the clock.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

realtimeclock→set_unixTime() YRealTimeClock
realtimeclock→setUnixTime()
realtimeclock.set_unixTime()

Changes the current time.

int **set_unixTime**(long **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() YRealTimeClock
realtimeclock→setUserData()
realtimeclock.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

realtimeclock→set_utcOffset() YRealTimeClock
realtimeclock→setUtcOffset()
realtimeclock.set_utcOffset()

Changes the number of seconds between current time and UTC time (time zone).

int **set_utcOffset**(int **newval**)

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time will automatically be updated according to the selected time zone.

Parameters :

newval an integer corresponding to the number of seconds between current time and UTC time (time zone)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.34. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_refframe.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YRefFrame = yoctolib.YRefFrame;
php     require_once('yocto_refframe.php');
cpp     #include "yocto_refframe.h"
m       #import "yocto_refframe.h"
pas     uses yocto_refframe;
vb      yocto_refframe.vb
cs      yocto_refframe.cs
java    import com.yoctopuce.YoctoAPI.YRefFrame;
py      from yocto_refframe import *
```

Global functions
yFindRefFrame(**func**) Retrieves a reference frame for a given identifier.
yFirstRefFrame() Starts the enumeration of reference frames currently accessible.
YRefFrame methods
refframe→cancel3DCalibration() Aborts the sensors tridimensional calibration process et restores normal settings.
refframe→describe() Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
refframe→get_3DCalibrationHint() Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.
refframe→get_3DCalibrationLogMsg() Returns the latest log message from the calibration process.
refframe→get_3DCalibrationProgress() Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.
refframe→get_3DCalibrationStage() Returns index of the current stage of the calibration initiated with method `start3DCalibration`.
refframe→get_3DCalibrationStageProgress() Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.
refframe→get_advertisedValue() Returns the current value of the reference frame (no more than 6 characters).
refframe→get_bearing() Returns the reference bearing used by the compass.

reframe→get_errorMessage()

Returns the error message of the latest error with the reference frame.

reframe→get_errorType()

Returns the numerical error code of the latest error with the reference frame.

reframe→get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

reframe→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

reframe→get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

reframe→get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

reframe→get_logicalName()

Returns the logical name of the reference frame.

reframe→get_module()

Gets the `YModule` object for the device on which the function is located.

reframe→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

reframe→get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

reframe→isOnline()

Checks if the reference frame is currently reachable, without raising any error.

reframe→isOnline_async(**callback**, **context**)

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reframe→load(**msValidity**)

Preloads the reference frame cache with a specified validity duration.

reframe→load_async(**msValidity**, **callback**, **context**)

Preloads the reference frame cache with a specified validity duration (asynchronous version).

reframe→more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

reframe→nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

reframe→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

reframe→save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

reframe→set_bearing(**newval**)

Changes the reference bearing used by the compass.

reframe→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() YRefFrame yFindRefFrame()YRefFrame.FindRefFrame()

Retrieves a reference frame for a given identifier.

YRefFrame **FindRefFrame**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the reference frame

Returns :

a `YRefFrame` object allowing you to drive the reference frame.

YRefFrame.FirstRefFrame() YRefFrame yFirstRefFrame()YRefFrame.FirstRefFrame()

Starts the enumeration of reference frames currently accessible.

YRefFrame **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() YRefFrame
refframe.cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

int **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`.

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 reference frame (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

refframe→get_3DCalibrationHint() YRefFrame
refframe→3DCalibrationHint()
refframe.get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

String **get_3DCalibrationHint**( )

Returns :
a character string.

refframe→get_3DCalibrationLogMsg() YRefFrame
refframe→3DCalibrationLogMsg()
`refframe.get_3DCalibrationLogMsg()`

Returns the latest log message from the calibration process.

String **get_3DCalibrationLogMsg**( )

When no new message is available, returns an empty string.

Returns :
a character string.

refframe→get_3DCalibrationProgress() YRefFrame
refframe→3DCalibrationProgress()
refframe.get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

int **get_3DCalibrationProgress**( )

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_3DCalibrationStage() YRefFrame
refframe→3DCalibrationStage()
refframe.get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

int **get_3DCalibrationStage**( )

Returns :
an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress() YRefFrame
refframe→3DCalibrationStageProgress()
refframe.get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

int **get_3DCalibrationStageProgress**( )

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue() YRefFrame
refframe→advertisedValue()
refframe.get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the reference frame (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

refframe→get_bearing() YRefFrame
refframe→bearing()`refframe.get_bearing()`

Returns the reference bearing used by the compass.

double **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.get_errorMessage()

Returns the error message of the latest error with the reference frame.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the reference frame object

refframe→get_errorType() YRefFrame
refframe→errorType()`refframe.get_errorType()`

Returns the numerical error code of the latest error with the reference frame.

```
int 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

reframe→get_friendlyName() YRefFrame
reframe→friendlyName()
refframe.get_friendlyName()

Returns a global identifier of the reference frame in the format MODULE_NAME.FUNCTION_NAME.

String **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() YRefFrame
refframe→functionDescriptor()
refframe.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

refframe→get_functionId() YRefFrame
refframe→functionId()
refframe.get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the reference frame (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

refframe→get_hardwareId() YRefFrame
refframe→hardwareId()
refframe.get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the reference frame (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

refframe→get_logicalName() YRefFrame
refframe→logicalName()
refframe.get_logicalName()

Returns the logical name of the reference frame.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the reference frame. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

refframe→get_module() YRefFrame
refframe→module()refframe.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

refframe→get_mountOrientation() YRefFrame
refframe→mountOrientation()
refframe.get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

MOUNTORIENTATION **get_mountOrientation**( )

Returns :

a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

refframe→get_mountPosition() YRefFrame
refframe→mountPosition()
refframe.get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

MOUNTPOSITION **get_mountPosition**( )

Returns :

a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.

refframe→get_userData() YRefFrame
refframe→userData()`refframe.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

Returns :

`true` if the reference frame can be reached, and `false` otherwise

refframe→load()refframe.load() YRefFrame

Preloads the reference frame cache with a specified validity duration.

int **load**(long **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→more3DCalibration() YRefFrame
refframe.more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

int **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() YRefFrame
refframe.nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

YRefFrame **nextRefFrame**( )

Returns :

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a `null` pointer if there are no more reference frames to enumerate.

refframe→registerValueCallback() YRefFrame
refframe.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

refframe→save3DCalibration() YRefFrame
refframe.save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

int **save3DCalibration**( )

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

refframe→set_bearing() YRefFrame
refframe→setBearing()`refframe.set_bearing()`

Changes the reference bearing used by the compass.

int **set_bearing**(double **newval**)

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→set_logicalName() YRefFrame
refframe→setLogicalName()
refframe.set_logicalName()

Changes the logical name of the reference frame.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the reference frame.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

refframe→set_mountPosition() YRefFrame
refframe→setMountPosition()
refframe.set_mountPosition()

Changes the compass and tilt sensor frame of reference.

```
int set_mountPosition( MOUNTPOSITION position,
                       MOUNTORIENTATION orientation)
```

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

Parameters :

position a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

orientation a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the saveToFlash() method of the module if the modification must be kept.

refframe→set_userData() YRefFrame
refframe→setUserData()`refframe.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

refframe→start3DCalibration() YRefFrame
refframe.start3DCalibration()

Initiates the sensors tridimensional calibration process.

int **start3DCalibration**( )

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

3.35. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_relay.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YRelay = yoctolib.YRelay;
php     require_once('yocto_relay.php');
cpp     #include "yocto_relay.h"
m       #import "yocto_relay.h"
pas     uses yocto_relay;
vb      yocto_relay.vb
cs      yocto_relay.cs
java    import com.yoctopuce.YoctoAPI.YRelay;
py      from yocto_relay import *
```

Global functions
yFindRelay(**func**) Retrieves a relay for a given identifier.
yFirstRelay() Starts the enumeration of relays currently accessible.
YRelay methods
relay→delayedPulse(**ms_delay**, **ms_duration**) Schedules a pulse.
relay→describe() Returns a short text that describes unambiguously the instance of the relay in the form TYPE(NAME)=SERIAL.FUNCTIONID.
relay→get_advertisedValue() Returns the current value of the relay (no more than 6 characters).
relay→get_countdown() Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.
relay→get_errorMessage() Returns the error message of the latest error with the relay.
relay→get_errorType() Returns the numerical error code of the latest error with the relay.
relay→get_friendlyName() Returns a global identifier of the relay in the format MODULE_NAME.FUNCTION_NAME.
relay→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
relay→get_functionId() Returns the hardware identifier of the relay, without reference to the module.
relay→get_hardwareId()

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

relay→get_logicalName()

Returns the logical name of the relay.

relay→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

relay→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

relay→get_module()

Gets the `YModule` object for the device on which the function is located.

relay→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

relay→get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

relay→get_pulseTimer()

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

relay→get_state()

Returns the state of the relays (A for the idle position, B for the active position).

relay→get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

relay→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

relay→isOnline()

Checks if the relay is currently reachable, without raising any error.

relay→isOnline_async(**callback**, **context**)

Checks if the relay is currently reachable, without raising any error (asynchronous version).

relay→load(**msValidity**)

Preloads the relay cache with a specified validity duration.

relay→load_async(**msValidity**, **callback**, **context**)

Preloads the relay cache with a specified validity duration (asynchronous version).

relay→nextRelay()

Continues the enumeration of relays started using `yFirstRelay()`.

relay→pulse(**ms_duration**)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

relay→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

relay→set_logicalName(**newval**)

Changes the logical name of the relay.

relay→set_maxTimeOnStateA(**newval**)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

relay→set_maxTimeOnStateB(**newval**)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

relay→set_output(**newval**)

Changes the output state of the relays, when used as a simple switch (single throw).

relay→set_state(**newval**)

Changes the state of the relays (A for the idle position, B for the active position).

relay→set_stateAtPowerOn(**newval**)

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

relay→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

relay→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRelay.FindRelay() YRelay
yFindRelay()YRelay.FindRelay()

Retrieves a relay for a given identifier.

YRelay **FindRelay**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the relay is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRelay.isOnline()` to test if the relay is indeed online at a given time. In case of ambiguity when looking for a relay by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the relay

Returns :

a `YRelay` object allowing you to drive the relay.

YRelay.FirstRelay() YRelay yFirstRelay()YRelay.FirstRelay()

Starts the enumeration of relays currently accessible.

YRelay **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.

int **delayedPulse**(int **ms_delay**, int **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.

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 relay (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

relay→get_advertisedValue() YRelay
relay→advertisedValue()
`relay.get_advertisedValue()`

Returns the current value of the relay (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the relay (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

relay→get_countdown() YRelay
relay→countdown()`relay.get_countdown()`

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

long **get_countdown**( )

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

relay→get_errorMessage() YRelay
relay→errorMessage()`relay.get_errorMessage()`

Returns the error message of the latest error with the relay.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the relay object

relay→get_errorType() YRelay
relay→errorType()`relay.get_errorType()`

Returns the numerical error code of the latest error with the relay.

int **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() YRelay
relay→friendlyName()`relay.get_friendlyName()`

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

String **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() YRelay
relay→functionDescriptor()
`relay.get_functionDescriptor()`

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

relay→get_functionId() YRelay
relay→functionId()`relay.get_functionId()`

Returns the hardware identifier of the relay, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the relay (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

relay→get_hardwareId() YRelay
relay→hardwareId()`relay.get_hardwareId()`

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the relay. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the relay (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

relay→get_logicalName() YRelay
relay→logicalName()`relay.get_logicalName()`

Returns the logical name of the relay.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the relay. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

relay→get_maxTimeOnStateA() YRelay
relay→maxTimeOnStateA()
relay.get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

long **get_maxTimeOnStateA**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.

relay→get_maxTimeOnStateB() YRelay
relay→maxTimeOnStateB()
relay.get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

long **get_maxTimeOnStateB**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEB_INVALID.

relay→get_module() YRelay
relay→module()`relay.get_module()`

Gets the `YModule` object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

Returns :
an instance of `YModule`

relay→get_output() YRelay
relay→output()relay.get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

int **get_output**( )

Returns :

either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.

relay→get_pulseTimer() YRelay
relay→pulseTimer()`relay.get_pulseTimer()`

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

long **get_pulseTimer**( )

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

relay→get_state() YRelay
relay→state()`relay.get_state()`

Returns the state of the relays (A for the idle position, B for the active position).

int **get_state**( )

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

relay→get_stateAtPowerOn() YRelay
relay→stateAtPowerOn()
relay.get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

int **get_stateAtPowerOn**( )

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

relay→get_userData() YRelay
relay→userData()`relay.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the relay.

Returns :

`true` if the relay can be reached, and `false` otherwise

relay→load()`relay.load()` YRelay

Preloads the relay cache with a specified validity duration.

int **load**(long **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→nextRelay()`relay.nextRelay()` YRelay

Continues the enumeration of relays started using `yFirstRelay()`.

YRelay **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).

int **pulse**(int **ms_duration**)

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→registerValueCallback() YRelay
`relay.registerValueCallback()`

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

relay→set_logicalName() YRelay
relay→setLogicalName()
relay.set_logicalName()

Changes the logical name of the relay.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the relay.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateA() YRelay
relay→setMaxTimeOnStateA()
relay.set_maxTimeOnStateA()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

int **set_maxTimeOnStateA**(long **newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateB() YRelay
relay→setMaxTimeOnStateB()
relay.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
int set_maxTimeOnStateB( long newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_output() YRelay
relay→setOutput()relay.set_output()

Changes the output state of the relays, when used as a simple switch (single throw).

int **set_output**(int **newval**)

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_state() YRelay
relay→setState()`relay.set_state()`

Changes the state of the relays (A for the idle position, B for the active position).

int **set_state**(int **newval**)

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_stateAtPowerOn() YRelay
relay→setStateAtPowerOn()
relay.set_stateAtPowerOn()

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

int **set_stateAtPowerOn**(int **newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_userData() YRelay
relay→setUserData()`relay.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.36. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
```

Global functions

yFindSensor(**func**)

Retrieves a sensor for a given identifier.

yFirstSensor()

Starts the enumeration of sensors currently accessible.

YSensor methods

sensor→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

sensor→describe()

Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

sensor→get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

sensor→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

sensor→get_currentValue()

Returns the current value of the measure.

sensor→get_errorMessage()

Returns the error message of the latest error with the sensor.

sensor→get_errorType()

Returns the numerical error code of the latest error with the sensor.

sensor→get_friendlyName()

Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.

sensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

sensor→get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

sensor→get_hardwareId()

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

sensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

sensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

sensor→get_logicalName()

Returns the logical name of the sensor.

sensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

sensor→get_module()

Gets the `YModule` object for the device on which the function is located.

sensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

sensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

sensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

sensor→get_resolution()

Returns the resolution of the measured values.

sensor→get_unit()

Returns the measuring unit for the measure.

sensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

sensor→isOnline()

Checks if the sensor is currently reachable, without raising any error.

sensor→isOnline_async(**callback**, **context**)

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

sensor→load(**msValidity**)

Preloads the sensor cache with a specified validity duration.

sensor→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

sensor→load_async(**msValidity**, **callback**, **context**)

Preloads the sensor cache with a specified validity duration (asynchronous version).

sensor→nextSensor()

Continues the enumeration of sensors started using `yFirstSensor()`.

sensor→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

sensor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

sensor→set_highestValue(**newval**)

Changes the recorded maximal value observed.

sensor→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

sensor→set_logicalName(**newval**)

Changes the logical name of the sensor.

sensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

sensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

sensor→set_resolution(newval)

Changes the resolution of the measured physical values.

sensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

sensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YSensor.FindSensor() YSensor yFindSensor()YSensor.FindSensor()

Retrieves a sensor for a given identifier.

YSensor **FindSensor**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the sensor

Returns :

a `YSensor` object allowing you to drive the sensor.

YSensor.FirstSensor() YSensor yFirstSensor()YSensor.FirstSensor()

Starts the enumeration of sensors currently accessible.

YSensor **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() YSensor
sensor.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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 sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

sensor→get_advertisedValue() YSensor
sensor→advertisedValue()
sensor.get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

sensor→get_currentRawValue() YSensor
sensor→currentRawValue()
sensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YSensor
sensor→currentValue()
sensor.get_currentValue()

Returns the current value of the measure.

double **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() YSensor
sensor→errorMessage()
sensor.get_errorMessage()

Returns the error message of the latest error with the sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the sensor object

sensor→get_errorType() YSensor
sensor→errorType()sensor.get_errorType()

Returns the numerical error code of the latest error with the sensor.

```
int 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.get_friendlyName()

Returns a global identifier of the sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YSensor
sensor→functionDescriptor()
sensor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

sensor→get_functionId() YSensor
sensor→functionId()sensor.get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

sensor→get_hardwareId() YSensor
sensor→hardwareId()sensor.get_hardwareId()

Returns the unique hardware identifier of the sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

sensor→get_highestValue() YSensor
sensor→highestValue()
sensor.get_highestValue()

Returns the maximal value observed for the measure since the device was started.

double **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

sensor→get_logFrequency() YSensor
sensor→logFrequency()
sensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

sensor→get_logicalName() YSensor
sensor→logicalName()sensor.get_logicalName()

Returns the logical name of the sensor.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

sensor→get_lowestValue() YSensor
sensor→lowestValue()sensor.get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

double **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

sensor→get_module() YSensor
sensor→module()sensor.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

sensor→get_recordedData() YSensor
sensor→recordedData()
sensor.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

sensor→get_reportFrequency() YSensor
sensor→reportFrequency()
sensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

sensor→get_resolution() YSensor
sensor→resolution()sensor.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

sensor→get_unit() YSensor
sensor→unit()sensor.get_unit()

Returns the measuring unit for the measure.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns Y_UNIT_INVALID.

sensor→get_userData() YSensor
sensor→userData()sensor.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the sensor.

Returns :
`true` if the sensor can be reached, and `false` otherwise

sensor→load()sensor.load() YSensor

Preloads the sensor cache with a specified validity duration.

int **load**(long **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() YSensor
sensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→nextSensor()sensor.nextSensor() YSensor

Continues the enumeration of sensors started using yFirstSensor().

YSensor **nextSensor**( )

Returns :

a pointer to a YSensor object, corresponding to a sensor currently online, or a null pointer if there are no more sensors to enumerate.

sensor→registerTimedReportCallback() YSensor
sensor.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

sensor→registerValueCallback() YSensor
sensor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

sensor→set_highestValue() YSensor
sensor→setHighestValue()
sensor.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_logFrequency() YSensor
sensor→setLogFrequency()
sensor.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_logicalName() YSensor
sensor→setLogicalName()
sensor.set_logicalName()

Changes the logical name of the sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

sensor→set_lowestValue() YSensor
sensor→setLowestValue()
sensor.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_reportFrequency() YSensor
sensor→setReportFrequency()
sensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_resolution() YSensor
sensor→setResolution()sensor.set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_userData() YSensor
sensor→setUserData()sensor.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.37. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_servo.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YServo = yoctolib.YServo;
php     require_once('yocto_servo.php');
cpp     #include "yocto_servo.h"
m       #import "yocto_servo.h"
pas     uses yocto_servo;
vb      yocto_servo.vb
cs      yocto_servo.cs
java    import com.yoctopuce.YoctoAPI.YServo;
py      from yocto_servo import *
```

Global functions
yFindServo(**func**) Retrieves a servo for a given identifier.
yFirstServo() Starts the enumeration of servos currently accessible.
YServo methods
servo→describe() Returns a short text that describes unambiguously the instance of the servo in the form TYPE(NAME)=SERIAL.FUNCTIONID.
servo→get_advertisedValue() Returns the current value of the servo (no more than 6 characters).
servo→get_enabled() Returns the state of the servos.
servo→get_enabledAtPowerOn() Returns the servo signal generator state at power up.
servo→get_errorMessage() Returns the error message of the latest error with the servo.
servo→get_errorType() Returns the numerical error code of the latest error with the servo.
servo→get_friendlyName() Returns a global identifier of the servo in the format MODULE_NAME.FUNCTION_NAME.
servo→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
servo→get_functionId() Returns the hardware identifier of the servo, without reference to the module.
servo→get_hardwareId() Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.
servo→get_logicalName() Returns the logical name of the servo.

servo→get_module()

Gets the `YModule` object for the device on which the function is located.

servo→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_positionAtPowerOn()

Returns the servo position at device power up.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

servo→isOnline()

Checks if the servo is currently reachable, without raising any error.

servo→isOnline_async(**callback**, **context**)

Checks if the servo is currently reachable, without raising any error (asynchronous version).

servo→load(**msValidity**)

Preloads the servo cache with a specified validity duration.

servo→load_async(**msValidity**, **callback**, **context**)

Preloads the servo cache with a specified validity duration (asynchronous version).

servo→move(**target**, **ms_duration**)

Performs a smooth move at constant speed toward a given position.

servo→nextServo()

Continues the enumeration of servos started using `yFirstServo()`.

servo→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

servo→set_enabled(**newval**)

Stops or starts the servo.

servo→set_enabledAtPowerOn(**newval**)

Configure the servo signal generator state at power up.

servo→set_logicalName(**newval**)

Changes the logical name of the servo.

servo→set_neutral(**newval**)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→set_position(**newval**)

Changes immediately the servo driving position.

servo→set_positionAtPowerOn(**newval**)

Configure the servo position at device power up.

servo→set_range(**newval**)

Changes the range of use of the servo, specified in per cents.

servo→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

servo→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YServo.FindServo() YServo yFindServo()YServo.FindServo()

Retrieves a servo for a given identifier.

YServo **FindServo**(String **func**)

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() YServo yFirstServo()YServo.FirstServo()

Starts the enumeration of servos currently accessible.

```
YServo 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.

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 servo (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

servo→get_advertisedValue() YServo
servo→advertisedValue()
servo.get_advertisedValue()

Returns the current value of the servo (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the servo (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

servo→get_enabled() YServo
servo→enabled()servo.get_enabled()

Returns the state of the servos.

int **get_enabled**( )

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the servos

On failure, throws an exception or returns Y_ENABLED_INVALID.

servo→get_enabledAtPowerOn() YServo
servo→enabledAtPowerOn()
servo.get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

int **get_enabledAtPowerOn**( )

Returns :

either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the servo signal generator state at power up

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.

servo→get_errorMessage() YServo
servo→errorMessage()servo.get_errorMessage()

Returns the error message of the latest error with the servo.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the servo object

servo→get_errorType() YServo
servo→errorType()servo.get_errorType()

Returns the numerical error code of the latest error with the servo.

```
int 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() YServo
servo→friendlyName()servo.get_friendlyName()

Returns a global identifier of the servo in the format MODULE_NAME.FUNCTION_NAME.

String **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() YServo
servo→functionDescriptor()
servo.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

servo→get_functionId() YServo
servo→functionId()servo.get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the servo (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

servo→get_hardwareId() YServo
servo→hardwareId()servo.get_hardwareId()

Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the servo. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the servo (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

servo→get_logicalName() YServo
servo→logicalName()`servo.get_logicalName()`

Returns the logical name of the servo.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the servo. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

servo→get_module() YServo
servo→module()servo.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

servo→get_neutral() YServo
servo→neutral()servo.get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

int **get_neutral**( )

Returns :
an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns Y_NEUTRAL_INVALID.

servo→get_position() YServo
servo→position()servo.get_position()

Returns the current servo position.

int **get_position**( )

Returns :
an integer corresponding to the current servo position

On failure, throws an exception or returns Y_POSITION_INVALID.

servo→get_positionAtPowerOn() YServo
servo→positionAtPowerOn()
servo.get_positionAtPowerOn()

Returns the servo position at device power up.

int **get_positionAtPowerOn**( )

Returns :
an integer corresponding to the servo position at device power up

On failure, throws an exception or returns Y_POSITIONATPOWERON_INVALID.

servo→get_range() YServo
servo→range()servo.get_range()

Returns the current range of use of the servo.

int **get_range**( )

Returns :
an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns Y_RANGE_INVALID.

servo→get_userData() YServo
servo→userData()servo.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

Returns :

`true` if the servo can be reached, and `false` otherwise

servo→load()servo.load() YServo

Preloads the servo cache with a specified validity duration.

int **load**(long **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→move()servo.move() YServo

Performs a smooth move at constant speed toward a given position.

```
int move( int target, int 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().

YServo **nextServo**( )

Returns :

a pointer to a YServo object, corresponding to a servo currently online, or a null pointer if there are no more servos to enumerate.

servo→registerValueCallback() YServo
servo.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

servo→set_enabled() YServo
servo→setEnabled()servo.set_enabled()

Stops or starts the servo.

int **set_enabled**(int **newval**)

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_enabledAtPowerOn() YServo
servo→setEnabledAtPowerOn()
servo.set_enabledAtPowerOn()

Configure the servo signal generator state at power up.

int **set_enabledAtPowerOn**(int **newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_logicalName() YServo
servo→setLogicalName()
servo.set_logicalName()

Changes the logical name of the servo.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the servo.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

servo→set_neutral() YServo
servo→setNeutral()`servo.set_neutral()`

Changes the duration of the pulse corresponding to the neutral position of the servo.

int **set_neutral**(int **newval**)

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_position() YServo
servo→setPosition()servo.set_position()

Changes immediately the servo driving position.

int **set_position**(int **newval**)

Parameters :

newval an integer corresponding to immediately the servo driving position

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_positionAtPowerOn() YServo
servo→setPositionAtPowerOn()
servo.set_positionAtPowerOn()

Configure the servo position at device power up.

int **set_positionAtPowerOn**(int **newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_range() YServo
servo→setRange()servo.set_range()

Changes the range of use of the servo, specified in per cents.

int **set_range**(int **newval**)

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms], When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the range of use of the servo, specified in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_userData() YServo
servo→setUserData()`servo.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.38. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_temperature.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YTemperature = yoctolib.YTemperature;
php     require_once('yocto_temperature.php');
cpp     #include "yocto_temperature.h"
m       #import "yocto_temperature.h"
pas     uses yocto_temperature;
vb      yocto_temperature.vb
cs      yocto_temperature.cs
java    import com.yoctopuce.YoctoAPI.YTemperature;
py      from yocto_temperature import *
```

Global functions
yFindTemperature(**func**) Retrieves a temperature sensor for a given identifier.
yFirstTemperature() Starts the enumeration of temperature sensors currently accessible.
`YTemperature` methods
temperature→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
temperature→describe() Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
temperature→get_advertisedValue() Returns the current value of the temperature sensor (no more than 6 characters).
temperature→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
temperature→get_currentValue() Returns the current value of the temperature.
temperature→get_errorMessage() Returns the error message of the latest error with the temperature sensor.
temperature→get_errorType() Returns the numerical error code of the latest error with the temperature sensor.
temperature→get_friendlyName() Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.
temperature→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
temperature→get_functionId() Returns the hardware identifier of the temperature sensor, without reference to the module.
temperature→get_hardwareId() Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

temperature→get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

temperature→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

temperature→get_logicalName()

Returns the logical name of the temperature sensor.

temperature→get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

temperature→get_module()

Gets the `YModule` object for the device on which the function is located.

temperature→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

temperature→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

temperature→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

temperature→get_resolution()

Returns the resolution of the measured values.

temperature→get_sensorType()

Returns the temperature sensor type.

temperature→get_unit()

Returns the measuring unit for the temperature.

temperature→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

temperature→isOnline()

Checks if the temperature sensor is currently reachable, without raising any error.

temperature→isOnline_async(callback, context)

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

temperature→load(msValidity)

Preloads the temperature sensor cache with a specified validity duration.

temperature→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

temperature→load_async(msValidity, callback, context)

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

temperature→nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

temperature→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

temperature→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

temperature→set_highestValue(newval)

Changes the recorded maximal value observed.

temperature→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

temperature→set_logicalName(newval)

Changes the logical name of the temperature sensor.

temperature→set_lowestValue(newval)

Changes the recorded minimal value observed.

temperature→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

temperature→set_resolution(newval)

Changes the resolution of the measured physical values.

temperature→set_sensorType(newval)

Modify the temperature sensor type.

temperature→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

temperature→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTemperature.FindTemperature() YTemperature
yFindTemperature()
YTemperature.FindTemperature()

Retrieves a temperature sensor for a given identifier.

```
YTemperature FindTemperature( String func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the temperature sensor

Returns :

a `YTemperature` object allowing you to drive the temperature sensor.

YTemperature.FirstTemperature() YTemperature yFirstTemperature() YTemperature.FirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

YTemperature **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() YTemperature
temperature.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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`.

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 temperature sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

temperature→get_advertisedValue() YTemperature
temperature→advertisedValue()
`temperature.get_advertisedValue()`

Returns the current value of the temperature sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the temperature sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

temperature→get_currentRawValue() YTemperature
temperature→currentRawValue()
temperature.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YTemperature
temperature→currentValue()
temperature.get_currentValue()

Returns the current value of the temperature.

double **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() YTemperature
temperature→errorMessage()
temperature.get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the temperature sensor object

temperature→get_errorType() YTemperature
temperature→errorType()
temperature.get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

```
int 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() YTemperature
temperature→friendlyName()
temperature.get_friendlyName()

Returns a global identifier of the temperature sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YTemperature
temperature→functionDescriptor()
temperature.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

temperature→get_functionId() YTemperature
temperature→functionId()
`temperature.get_functionId()`

Returns the hardware identifier of the temperature sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the temperature sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

temperature→get_hardwareId() YTemperature
temperature→hardwareId()
temperature.get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the temperature sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the temperature sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

temperature→get_highestValue() YTemperature
temperature→highestValue()
temperature.get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

double **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

temperature→get_logFrequency() YTemperature
temperature→logFrequency()
temperature.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

temperature→get_logicalName() YTemperature
temperature→logicalName()
temperature.get_logicalName()

Returns the logical name of the temperature sensor.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the temperature sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

temperature→get_lowestValue() YTemperature
temperature→lowestValue()
temperature.get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

double **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

temperature→get_module() YTemperature
temperature→module()
temperature.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

temperature→get_recordedData() YTemperature
temperature→recordedData()
temperature.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

temperature→get_reportFrequency() YTemperature
temperature→reportFrequency()
`temperature.get_reportFrequency()`

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

temperature→get_resolution() YTemperature
temperature→resolution()
`temperature.get_resolution()`

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

temperature→get_sensorType() YTemperature
temperature→sensorType()
temperature.get_sensorType()

Returns the temperature sensor type.

int **get_sensorType**( )

Returns :

a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES corresponding to the temperature sensor type

On failure, throws an exception or returns Y_SENSORTYPE_INVALID.

temperature→get_unit() YTemperature
temperature→unit()`temperature.get_unit()`

Returns the measuring unit for the temperature.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the temperature

On failure, throws an exception or returns `Y_UNIT_INVALID`.

temperature→get_userData() YTemperature
temperature→userData()
temperature.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

Returns :

`true` if the temperature sensor can be reached, and `false` otherwise

temperature→load()`temperature.load()` YTemperature

Preloads the temperature sensor cache with a specified validity duration.

int **load**(long **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() YTemperature
temperature.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→nextTemperature() YTemperature
temperature.nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

YTemperature **nextTemperature**( )

Returns :

a pointer to a `YTemperature` object, corresponding to a temperature sensor currently online, or a `null` pointer if there are no more temperature sensors to enumerate.

temperature→registerTimedReportCallback() YTemperature
temperature.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

temperature→registerValueCallback() YTemperature
temperature.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

temperature→set_highestValue() YTemperature
temperature→setHighestValue()
temperature.set_highestValue()

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logFrequency() YTemperature
temperature→setLogFrequency()
`temperature.set_logFrequency()`

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logicalName() YTemperature
temperature→setLogicalName()
temperature.set_logicalName()

Changes the logical name of the temperature sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the temperature sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

temperature→set_lowestValue() YTemperature
temperature→setLowestValue()
temperature.set_lowestValue()

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_reportFrequency() YTemperature
temperature→setReportFrequency()
temperature.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_resolution() YTemperature
temperature→setResolution()
temperature.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_sensorType() YTemperature
temperature→setSensorType()
temperature.set_sensorType()

Modify the temperature sensor type.

int **set_sensorType**(int **newval**)

This function is used to to define the type of thermocouple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the saveToFlash() method of the module if the modification must be kept.

Parameters :

newval a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_userData() YTemperature
temperature→setUserData()
temperature.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.39. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_tilt.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YTilt = yoctolib.YTilt;
php     require_once('yocto_tilt.php');
cpp     #include "yocto_tilt.h"
m       #import "yocto_tilt.h"
pas     uses yocto_tilt;
vb      yocto_tilt.vb
cs      yocto_tilt.cs
java    import com.yoctopuce.YoctoAPI.YTilt;
py      from yocto_tilt import *
```

Global functions
yFindTilt(**func**) Retrieves a tilt sensor for a given identifier.
yFirstTilt() Starts the enumeration of tilt sensors currently accessible.
YTilt methods
tilt→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
tilt→describe() Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
tilt→get_advertisedValue() Returns the current value of the tilt sensor (no more than 6 characters).
tilt→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
tilt→get_currentValue() Returns the current value of the inclination.
tilt→get_errorMessage() Returns the error message of the latest error with the tilt sensor.
tilt→get_errorType() Returns the numerical error code of the latest error with the tilt sensor.
tilt→get_friendlyName() Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.
tilt→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
tilt→get_functionId() Returns the hardware identifier of the tilt sensor, without reference to the module.
tilt→get_hardwareId() Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

tilt→get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

tilt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

tilt→get_logicalName()

Returns the logical name of the tilt sensor.

tilt→get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

tilt→get_module()

Gets the `YModule` object for the device on which the function is located.

tilt→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

tilt→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

tilt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

tilt→get_resolution()

Returns the resolution of the measured values.

tilt→get_unit()

Returns the measuring unit for the inclination.

tilt→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

tilt→isOnline()

Checks if the tilt sensor is currently reachable, without raising any error.

tilt→isOnline_async(**callback**, **context**)

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

tilt→load(**msValidity**)

Preloads the tilt sensor cache with a specified validity duration.

tilt→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

tilt→load_async(**msValidity**, **callback**, **context**)

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

tilt→nextTilt()

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

tilt→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

tilt→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

tilt→set_highestValue(**newval**)

Changes the recorded maximal value observed.

tilt→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

tilt→set_logicalName(**newval**)

Changes the logical name of the tilt sensor.

tilt→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

tilt→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

tilt→set_resolution(**newval**)

Changes the resolution of the measured physical values.

tilt→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

tilt→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTilt.FindTilt() YTilt
yFindTilt()YTilt.FindTilt()

Retrieves a tilt sensor for a given identifier.

YTilt **FindTilt**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the tilt sensor

Returns :

a `YTilt` object allowing you to drive the tilt sensor.

YTilt.FirstTilt() YTilt yFirstTilt()YTilt.FirstTilt()

Starts the enumeration of tilt sensors currently accessible.

```
YTilt 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() YTilt
tilt.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                                                ArrayList<Double> 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.

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 tilt sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

tilt→get_advertisedValue() YTilt
tilt→advertisedValue()
tilt.get_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :
a string corresponding to the current value of the tilt sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

tilt→get_currentRawValue() YTilt
tilt→currentRawValue()
tilt.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YTilt
tilt→currentValue()`tilt.get_currentValue()`

Returns the current value of the inclination.

double **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() YTilt
tilt→errorMessage()`tilt.get_errorMessage()`

Returns the error message of the latest error with the tilt sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the tilt sensor object

tilt→get_errorType() YTilt
tilt→errorType()`tilt.get_errorType()`

Returns the numerical error code of the latest error with the tilt sensor.

int **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() YTilt
tilt→friendlyName()`tilt.get_friendlyName()`

Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

String **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() YTilt
tilt→functionDescriptor()
tilt.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

tilt→get_functionId() YTilt
tilt→functionId()`tilt.get_functionId()`

Returns the hardware identifier of the tilt sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the tilt sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

tilt→get_hardwareId() YTilt
tilt→hardwareId()`tilt.get_hardwareId()`

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the tilt sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the tilt sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

tilt→get_highestValue() YTilt
tilt→highestValue()`tilt.get_highestValue()`

Returns the maximal value observed for the inclination since the device was started.

double **get_highestValue**( )

Returns :
a floating point number corresponding to the maximal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

tilt→get_logFrequency() YTilt
tilt→logFrequency()`tilt.get_logFrequency()`

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

tilt→get_logicalName() YTilt
tilt→logicalName()`tilt.get_logicalName()`

Returns the logical name of the tilt sensor.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the tilt sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

tilt→get_lowestValue() YTilt
tilt→lowestValue()`tilt.get_lowestValue()`

Returns the minimal value observed for the inclination since the device was started.

double **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

tilt→get_module() YTilt
tilt→module()tilt.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

tilt→get_recordedData() YTilt
tilt→recordedData()`tilt.get_recordedData()`

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

tilt→get_reportFrequency() YTilt
tilt→reportFrequency()
tilt.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

tilt→get_resolution() YTilt
tilt→resolution()`tilt.get_resolution()`

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

tilt→get_unit() YTilt
tilt→unit()`tilt.get_unit()`

Returns the measuring unit for the inclination.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns `Y_UNIT_INVALID`.

tilt→get_userData() YTilt
tilt→userData()`tilt.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

Returns :

`true` if the tilt sensor can be reached, and `false` otherwise

tilt→load()`tilt.load()` YTilt

Preloads the tilt sensor cache with a specified validity duration.

int **load**(long **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() YTilt
`tilt.loadCalibrationPoints()`

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→nextTilt()`tilt.nextTilt()` YTilt

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

YTilt **nextTilt**( )

Returns :

a pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a `null` pointer if there are no more tilt sensors to enumerate.

tilt→registerTimedReportCallback() YTilt
`tilt.registerTimedReportCallback()`

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

tilt→registerValueCallback() YTilt
`tilt.registerValueCallback()`

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

tilt→set_highestValue() YTilt
tilt→setHighestValue()`tilt.set_highestValue()`

Changes the recorded maximal value observed.

int **set_highestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logFrequency() YTilt
tilt→setLogFrequency()`tilt.set_logFrequency()`

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logicalName() YTilt
tilt→setLogicalName()`tilt.set_logicalName()`

Changes the logical name of the tilt sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the tilt sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

tilt→set_lowestValue() YTilt
tilt→setLowestValue()`tilt.set_lowestValue()`

Changes the recorded minimal value observed.

int **set_lowestValue**(double **newval**)

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_reportFrequency() YTilt
tilt→setReportFrequency()
tilt.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_resolution() YTilt
tilt→setResolution()`tilt.set_resolution()`

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_userData() YTilt
tilt→setUserData()`tilt.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.40. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_voc.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YVoc = yoctolib.YVoc;
php     require_once('yocto_voc.php');
cpp     #include "yocto_voc.h"
m       #import "yocto_voc.h"
pas     uses yocto_voc;
vb      yocto_voc.vb
cs      yocto_voc.cs
java    import com.yoctopuce.YoctoAPI.YVoc;
py      from yocto_voc import *
```

Global functions

yFindVoc(**func**)

Retrieves a Volatile Organic Compound sensor for a given identifier.

yFirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

YVoc methods

voc→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voc→describe()

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

voc→get_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

voc→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

voc→get_currentValue()

Returns the current measure for the estimated VOC concentration.

voc→get_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

voc→get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

voc→get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

voc→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voc→get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

voc→get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

voc→get_highestValue()

Returns the maximal value observed for the estimated VOC concentration.

voc→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voc→get_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

voc→get_lowestValue()

Returns the minimal value observed for the estimated VOC concentration.

voc→get_module()

Gets the `YModule` object for the device on which the function is located.

voc→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voc→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voc→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voc→get_resolution()

Returns the resolution of the measured values.

voc→get_unit()

Returns the measuring unit for the estimated VOC concentration.

voc→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voc→isOnline()

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

voc→isOnline_async(**callback**, **context**)

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

voc→load(**msValidity**)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

voc→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voc→load_async(**msValidity**, **callback**, **context**)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

voc→nextVoc()

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

voc→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

voc→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

voc→set_highestValue(**newval**)

Changes the recorded maximal value observed for the estimated VOC concentration.

voc→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voc→set_logicalName(newval)

Changes the logical name of the Volatile Organic Compound sensor.

voc→set_lowestValue(newval)

Changes the recorded minimal value observed for the estimated VOC concentration.

voc→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voc→set_resolution(newval)

Changes the resolution of the measured physical values.

voc→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voc→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoc.FindVoc() YVoc yFindVoc()YVoc.FindVoc()

Retrieves a Volatile Organic Compound sensor for a given identifier.

YVoc **FindVoc**(String **func**)

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() YVoc yFirstVoc()YVoc.FirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

```
YVoc 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() YVoc
voc.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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 Volatile Organic Compound sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

voc→get_advertisedValue() YVoc
voc→advertisedValue()
voc.get_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

voc→get_currentRawValue() YVoc
voc→currentRawValue()
voc.get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **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() YVoc
voc→currentValue()voc.get_currentValue()

Returns the current measure for the estimated VOC concentration.

double **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() YVoc
voc→errorMessage()voc.get_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the Volatile Organic Compound sensor object

voc→get_errorType() YVoc
voc→errorType()voc.get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

```
int 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() YVoc
voc→friendlyName()voc.get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YVoc
voc→functionDescriptor()
voc.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voc→get_functionId() YVoc
voc→functionId()voc.get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the Volatile Organic Compound sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voc→get_hardwareId() YVoc
voc→hardwareId()voc.get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Volatile Organic Compound sensor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the Volatile Organic Compound sensor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

voc→get_highestValue() YVoc
voc→highestValue()voc.get_highestValue()

Returns the maximal value observed for the estimated VOC concentration.

double **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() YVoc
voc→logFrequency()voc.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

voc→get_logicalName() YVoc
voc→logicalName()voc.get_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the Volatile Organic Compound sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

voc→get_lowestValue() YVoc
voc→lowestValue()voc.get_lowestValue()

Returns the minimal value observed for the estimated VOC concentration.

double **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() YVoc
voc→module()voc.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

voc→get_recordedData() YVoc
voc→recordedData()voc.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voc→get_reportFrequency() YVoc
voc→reportFrequency()
voc.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

voc→get_resolution() YVoc
voc→resolution()voc.get_resolution()

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voc→get_unit() YVoc
voc→unit()voc.get_unit()

Returns the measuring unit for the estimated VOC concentration.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns Y_UNIT_INVALID.

voc→get_userData() YVoc
voc→userData()voc.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Volatile Organic Compound sensor.

Returns :

`true` if the Volatile Organic Compound sensor can be reached, and `false` otherwise

voc→load()voc.load() YVoc

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

int **load**(long **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() YVoc
voc.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→nextVoc()voc.nextVoc() YVoc

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

YVoc **nextVoc**( )

Returns :

a pointer to a `YVoc` object, corresponding to a Volatile Organic Compound sensor currently online, or a `null` pointer if there are no more Volatile Organic Compound sensors to enumerate.

voc→registerTimedReportCallback() YVoc
voc.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

voc→registerValueCallback() YVoc
voc.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voc→set_highestValue() YVoc
voc→setHighestValue()voc.set_highestValue()

Changes the recorded maximal value observed for the estimated VOC concentration.

int **set_highestValue**(double **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() YVoc
voc→setLogFrequency()voc.set_logFrequency()

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_logicalName() YVoc
voc→setLogicalName()voc.set_logicalName()

Changes the logical name of the Volatile Organic Compound sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Volatile Organic Compound sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

voc→set_lowestValue() YVoc
voc→setLowestValue()voc.set_lowestValue()

Changes the recorded minimal value observed for the estimated VOC concentration.

int **set_lowestValue**(double **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() YVoc
voc→setReportFrequency()
voc.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_resolution() YVoc
voc→setResolution()voc.set_resolution()

Changes the resolution of the measured physical values.

int **set_resolution**(double **newval**)

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_userData() YVoc
voc→setUserData()voc.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.41. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_voltage.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YVoltage = yoctolib.YVoltage;
php     require_once('yocto_voltage.php');
cpp     #include "yocto_voltage.h"
m       #import "yocto_voltage.h"
pas     uses yocto_voltage;
vb      yocto_voltage.vb
cs      yocto_voltage.cs
java    import com.yoctopuce.YoctoAPI.YVoltage;
py      from yocto_voltage import *
```

Global functions
yFindVoltage(**func**) Retrieves a voltage sensor for a given identifier.
yFirstVoltage() Starts the enumeration of voltage sensors currently accessible.
YVoltage methods
voltage→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
voltage→describe() Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
voltage→get_advertisedValue() Returns the current value of the voltage sensor (no more than 6 characters).
voltage→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
voltage→get_currentValue() Returns the current measure for the voltage.
voltage→get_errorMessage() Returns the error message of the latest error with the voltage sensor.
voltage→get_errorType() Returns the numerical error code of the latest error with the voltage sensor.
voltage→get_friendlyName() Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.
voltage→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
voltage→get_functionId() Returns the hardware identifier of the voltage sensor, without reference to the module.
voltage→get_hardwareId() Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

voltage→get_highestValue()

Returns the maximal value observed for the voltage.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage.

voltage→get_module()

Gets the `YModule` object for the device on which the function is located.

voltage→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(**callback**, **context**)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→load(**msValidity**)

Preloads the voltage sensor cache with a specified validity duration.

voltage→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voltage→load_async(**msValidity**, **callback**, **context**)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

voltage→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_highestValue(**newval**)

Changes the recorded maximal value observed pour the voltage.

voltage→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(**newval**)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed pour the voltage.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured values.

voltage→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltage.FindVoltage() YVoltage
yFindVoltage()YVoltage.FindVoltage()

Retrieves a voltage sensor for a given identifier.

YVoltage **FindVoltage**(String **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FirstVoltage() YVoltage yFirstVoltage()YVoltage.FirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

YVoltage **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() YVoltage
`voltage.calibrateFromPoints()`

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                         ArrayList<Double> 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.

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 voltage sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

voltage→get_advertisedValue() YVoltage
voltage→advertisedValue()
`voltage.get_advertisedValue()`

Returns the current value of the voltage sensor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the voltage sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

voltage→get_currentRawValue() YVoltage
voltage→currentRawValue()
voltage.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **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() YVoltage
voltage→currentValue()
`voltage.get_currentValue()`

Returns the current measure for the voltage.

double **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() YVoltage
voltage→errorMessage()
voltage.get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the voltage sensor object

voltage→get_errorType() YVoltage
voltage→errorType()`voltage.get_errorType()`

Returns the numerical error code of the latest error with the voltage sensor.

int **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() YVoltage
voltage→friendlyName()
voltage.get_friendlyName()

Returns a global identifier of the voltage sensor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YVoltage
voltage→functionDescriptor()
voltage.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voltage→get_functionId() YVoltage
voltage→functionId()`voltage.get_functionId()`

Returns the hardware identifier of the voltage sensor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the voltage sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voltage→get_hardwareId() YVoltage
voltage→hardwareId()`voltage.get_hardwareId()`

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the voltage sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

voltage→get_highestValue() YVoltage
voltage→highestValue()
voltage.get_highestValue()

Returns the maximal value observed for the voltage.

double **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() YVoltage
voltage→logFrequency()
`voltage.get_logFrequency()`

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String **get_logFrequency**( )

Returns :
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

voltage→get_logicalName() YVoltage
voltage→logicalName()
voltage.get_logicalName()

Returns the logical name of the voltage sensor.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the voltage sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

voltage→get_lowestValue() YVoltage
voltage→lowestValue()
voltage.get_lowestValue()

Returns the minimal value observed for the voltage.

double **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() YVoltage
voltage→module()`voltage.get_module()`

Gets the `YModule` object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :
an instance of `YModule`

voltage→get_recordedData() YVoltage
voltage→recordedData()
voltage.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(long **startTime**, long **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voltage→get_reportFrequency() YVoltage
voltage→reportFrequency()
`voltage.get_reportFrequency()`

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String **get_reportFrequency**( )

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

voltage→get_resolution() YVoltage
voltage→resolution()`voltage.get_resolution()`

Returns the resolution of the measured values.

double **get_resolution**( )

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voltage→get_unit() YVoltage
voltage→unit()`voltage.get_unit()`

Returns the measuring unit for the voltage.

String **get_unit**( )

Returns :
a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voltage→get_userData() YVoltage
voltage→userData()`voltage.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

Returns :

`true` if the voltage sensor can be reached, and `false` otherwise

voltage→load()`voltage.load()` YVoltage

Preloads the voltage sensor cache with a specified validity duration.

int **load**(long **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() YVoltage
voltage.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→nextVoltage()`voltage.nextVoltage()` YVoltage

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

YVoltage **nextVoltage**( )

Returns :

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a `null` pointer if there are no more voltage sensors to enumerate.

voltage→registerTimedReportCallback() YVoltage
voltage.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(TimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

voltage→registerValueCallback() YVoltage
voltage.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voltage→set_highestValue() YVoltage
voltage→setHighestValue()
voltage.set_highestValue()

Changes the recorded maximal value observed pour the voltage.

int **set_highestValue**(double **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() YVoltage
voltage→setLogFrequency()
`voltage.set_logFrequency()`

Changes the datalogger recording frequency for this function.

int **set_logFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_logicalName() YVoltage
voltage→setLogicalName()
voltage.set_logicalName()

Changes the logical name of the voltage sensor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

voltage→set_lowestValue() YVoltage
voltage→setLowestValue()
voltage.set_lowestValue()

Changes the recorded minimal value observed pour the voltage.

int **set_lowestValue**(double **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() YVoltage
voltage→setReportFrequency()
voltage.set_reportFrequency()

Changes the timed value notification frequency for this function.

int **set_reportFrequency**(String **newval**)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_resolution() YVoltage
voltage→setResolution()
voltage.set_resolution()

Changes the resolution of the measured values.

int **set_resolution**(double **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() YVoltage
voltage→setUserData()`voltage.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.42. 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 YVSource.FindVSource()YVSource.FindVSource()

Retrieves a voltage source for a given identifier.

YVSource **FindVSource**(String **func**)

yFindVSource() — YVSource.FindVSource()YVSource.FindVSource()

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 YVSource.FirstVSource() YVSource.FirstVSource()

Starts the enumeration of voltage sources currently accessible.

YVSource **FirstVSource**( )

yFirstVSource() — YVSource.FirstVSource()YVSource.FirstVSource()

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.

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() YVSource
vsource→advertisedValue()
vsource.get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

String **get_advertisedValue**( )

vsource→get_advertisedValue()
vsource→advertisedValue()vsource.get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

js	function **get_advertisedValue**( )
php	function **get_advertisedValue**( )
cpp	string **get_advertisedValue**( )
m	-(NSString*) advertisedValue
pas	function **get_advertisedValue**( ): string
vb	function **get_advertisedValue**( ) As String
cs	string **get_advertisedValue**( )
java	String **get_advertisedValue**( )
py	def **get_advertisedValue**( )
cmd	YVSource **target get_advertisedValue**

Returns :
a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

vsource→get_errorMessage() YVSource
vsource→errorMessage()
vsource.get_errorMessage()

Returns the error message of the latest error with this function.

String **get_errorMessage**( )

vsource→get_errorMessage()
vsource→errorMessage()vsource.get_errorMessage()

Returns the error message of the latest error with this function.

`js` function **get_errorMessage**( )
`php` function **get_errorMessage**( )
`cpp` string **get_errorMessage**( )
`m` -(NSString*) errorMessage
`pas` function **get_errorMessage**( ): string
`vb` function **get_errorMessage**( ) As String
`cs` string **get_errorMessage**( )
`java` String **get_errorMessage**( )
`py` def **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using this function object

vsource→get_errorType() YVSource
vsource→errorType()vsource.get_errorType()

Returns the numerical error code of the latest error with this function.

int **get_errorType**( )

vsource→get_errorType()
vsource→errorType()vsource.get_errorType()

Returns the numerical error code of the latest error with this function.

`js` function **get_errorType**( )
`php` function **get_errorType**( )
`cpp` YRETCODE **get_errorType**( )
`pas` function **get_errorType**( ): YRETCODE
`vb` function **get_errorType**( ) As YRETCODE
`cs` YRETCODE **get_errorType**( )
`java` int **get_errorType**( )
`py` def **get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occured while using this function object

vsource→get_extPowerFailure() YVSource
vsource→extPowerFailure()
vsource.get_extPowerFailure()

Returns true if external power supply voltage is too low.

int **get_extPowerFailure**( )

vsource→get_extPowerFailure()
vsource→extPowerFailure()vsource.get_extPowerFailure()

Returns true if external power supply voltage is too low.

js	function **get_extPowerFailure**( )
php	function **get_extPowerFailure**( )
cpp	Y_EXTPOWERFAILURE_enum **get_extPowerFailure**( )
m	-(Y_EXTPOWERFAILURE_enum) extPowerFailure
pas	function **get_extPowerFailure**( ): Integer
vb	function **get_extPowerFailure**( ) As Integer
cs	int **get_extPowerFailure**( )
java	int **get_extPowerFailure**( )
py	def **get_extPowerFailure**( )
cmd	YVSource **target get_extPowerFailure**

Returns :

either `Y_EXTPOWERFAILURE_FALSE` or `Y_EXTPOWERFAILURE_TRUE`, according to true if external power supply voltage is too low

On failure, throws an exception or returns `Y_EXTPOWERFAILURE_INVALID`.

vsource→get_failure() YVSource
vsource→failure()vsource.get_failure()

Returns true if the module is in failure mode.

```
int get_failure( )
```

vsource→get_failure()
vsource→failure()vsource.get_failure()

Returns true if the module is in failure mode.

```
js    function get_failure( )
php   function get_failure( )
cpp   Y_FAILURE_enum get_failure( )
m     -(Y_FAILURE_enum) failure
pas   function get_failure( ): Integer
vb    function get_failure( ) As Integer
cs    int get_failure( )
java  int get_failure( )
py    def get_failure( )
cmd   YVSource target get_failure
```

More information can be obtained by testing get_overheat, get_overcurrent etc... When a error condition is met, the output voltage is set to zéro and cannot be changed until the reset() function is called.

Returns :

either Y_FAILURE_FALSE or Y_FAILURE_TRUE, according to true if the module is in failure mode

On failure, throws an exception or returns Y_FAILURE_INVALID.

vsource→get_friendlyName() YVSource
vsource→friendlyName()
vsource.get_friendlyName()

Returns a global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

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() YVSource
vsource→functionDescriptor()
vsource.get_vsourceDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

vsource→get_functionDescriptor()
vsource→functionDescriptor()vsource.get_vsourceDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	function **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	def **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

vsource→get_functionId() YVSource
vsource→functionId()vsource.get_vsourceId()

Returns the hardware identifier of the function, without reference to the module.

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() YVSource
vsource→hardwareId()`vsource.get_hardwareId()`

Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

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() YVSource
vsource→logicalName()
vsource.get_logicalName()

Returns the logical name of the voltage source.

String **get_logicalName**( )

vsource→get_logicalName()
vsource→logicalName()vsource.get_logicalName()

Returns the logical name of the voltage source.

js	function **get_logicalName**( )
php	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	function **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
py	def **get_logicalName**( )
cmd	YVSource **target get_logicalName**

Returns :
a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

vsource→get_module() YVSource
vsource→module()vsource.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

vsource→get_module()
vsource→module()vsource.get_module()

Gets the YModule object for the device on which the function is located.

js	function **get_module**( )
php	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	function **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	def **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

vsource→get_overCurrent() YVSource
vsource→overCurrent()
vsource.get_overCurrent()

Returns true if the appliance connected to the device is too greedy .

int **get_overCurrent**( )

vsource→get_overCurrent()
vsource→overCurrent()vsource.get_overCurrent()

Returns true if the appliance connected to the device is too greedy .

```
js    function get_overCurrent( )
php   function get_overCurrent( )
cpp   Y_OVERCURRENT_enum get_overCurrent( )
m     -(Y_OVERCURRENT_enum) overCurrent
pas   function get_overCurrent( ): Integer
vb    function get_overCurrent( ) As Integer
cs    int get_overCurrent( )
java  int get_overCurrent( )
py    def get_overCurrent( )
cmd   YVSource target get_overCurrent
```

Returns :

either Y_OVERCURRENT_FALSE or Y_OVERCURRENT_TRUE, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns Y_OVERCURRENT_INVALID.

vsource→get_overHeat() YVSource
vsource→overHeat()vsource.get_overHeat()

Returns TRUE if the module is overheating.

int **get_overHeat**( )

vsource→get_overHeat()
vsource→overHeat()vsource.get_overHeat()

Returns TRUE if the module is overheating.

js	function **get_overHeat**( )
php	function **get_overHeat**( )
cpp	Y_OVERHEAT_enum **get_overHeat**( )
m	-(Y_OVERHEAT_enum) overHeat
pas	function **get_overHeat**( ): Integer
vb	function **get_overHeat**( ) As Integer
cs	int **get_overHeat**( )
java	int **get_overHeat**( )
py	def **get_overHeat**( )
cmd	YVSource **target get_overHeat**

Returns :

either Y_OVERHEAT_FALSE or Y_OVERHEAT_TRUE, according to TRUE if the module is overheating

On failure, throws an exception or returns Y_OVERHEAT_INVALID.

vsource→get_overLoad() YVSource
vsource→overLoad()vsource.get_overLoad()

Returns true if the device is not able to maintaint the requested voltage output .

int **get_overLoad**( )

vsource→get_overLoad()
vsource→overLoad()vsource.get_overLoad()

Returns true if the device is not able to maintaint the requested voltage output .

js	function **get_overLoad**( )
php	function **get_overLoad**( )
cpp	Y_OVERLOAD_enum **get_overLoad**( )
m	-(Y_OVERLOAD_enum) overLoad
pas	function **get_overLoad**( ): Integer
vb	function **get_overLoad**( ) As Integer
cs	int **get_overLoad**( )
java	int **get_overLoad**( )
py	def **get_overLoad**( )
cmd	YVSource **target get_overLoad**

Returns :

either Y_OVERLOAD_FALSE or Y_OVERLOAD_TRUE, according to true if the device is not able to maintaint the requested voltage output

On failure, throws an exception or returns Y_OVERLOAD_INVALID.

vsource→get_regulationFailure() YVSource
vsource→regulationFailure()
vsource.get_regulationFailure()

Returns true if the voltage output is too high regarding the requested voltage .

int **get_regulationFailure**( )

vsource→get_regulationFailure()
vsource→regulationFailure()vsource.get_regulationFailure()

Returns true if the voltage output is too high regarding the requested voltage .

js	function **get_regulationFailure**( )
php	function **get_regulationFailure**( )
cpp	Y_REGULATIONFAILURE_enum **get_regulationFailure**( )
m	-(Y_REGULATIONFAILURE_enum) regulationFailure
pas	function **get_regulationFailure**( ): Integer
vb	function **get_regulationFailure**( ) As Integer
cs	int **get_regulationFailure**( )
java	int **get_regulationFailure**( )
py	def **get_regulationFailure**( )
cmd	YVSource **target get_regulationFailure**

Returns :

either Y_REGULATIONFAILURE_FALSE or Y_REGULATIONFAILURE_TRUE, according to true if the voltage output is too high regarding the requested voltage

On failure, throws an exception or returns Y_REGULATIONFAILURE_INVALID.

vsource→get_unit() YVSource
vsource→unit()vsource.get_unit()

Returns the measuring unit for the voltage.

String **get_unit**( )

vsource→get_unit()
vsource→unit()vsource.get_unit()

Returns the measuring unit for the voltage.

```
js    function get_unit( )
php   function get_unit( )
cpp   string get_unit( )
m     -(NSString*) unit
pas   function get_unit( ): string
vb    function get_unit( ) As String
cs    string get_unit( )
java  String get_unit( )
py    def get_unit( )
cmd   YVSource target get_unit
```

Returns :

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

vsource→get_userData() YVSource
vsource→userData()vsource.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

vsource→get_userData()
vsource→userData()vsource.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

`js` function **get_userData**( )
`php` function **get_userData**( )
`cpp` void * **get_userData**( )
`m` -(void*) userData
`pas` function **get_userData**( ): Tobject
`vb` function **get_userData**( ) As Object
`cs` object **get_userData**( )
`java` Object **get_userData**( )
`py` def **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

vsource→get_voltage() YVSource
vsource→voltage()vsource.get_voltage()

Returns the voltage output command (mV)

int **get_voltage**( )

vsource→get_voltage()
vsource→voltage()vsource.get_voltage()

Returns the voltage output command (mV)

js	function **get_voltage**( )
php	function **get_voltage**( )
cpp	int **get_voltage**( )
m	-(int) voltage
pas	function **get_voltage**( ): LongInt
vb	function **get_voltage**( ) As Integer
cs	int **get_voltage**( )
java	int **get_voltage**( )
py	def **get_voltage**( )

Returns :
an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns Y_VOLTAGE_INVALID.

vsource→isOnline()`vsource.isOnline()` YVSource

Checks if the function is currently reachable, without raising any error.

boolean **isOnline**( )

vsource→isOnline()`vsource.isOnline()`

Checks if the function is currently reachable, without raising any error.

`js` function **isOnline**( )
`php` function **isOnline**( )
`cpp` bool **isOnline**( )
`m` -(BOOL) **isOnline**
`pas` function **isOnline**( ): boolean
`vb` function **isOnline**( ) As Boolean
`cs` bool **isOnline**( )
`java` boolean **isOnline**( )
`py` def **isOnline**( )

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :
`true` if the function can be reached, and `false` otherwise

vsource→load()vsource.load() YVSource

Preloads the function cache with a specified validity duration.

int **load**(long **msValidity**)

vsource→load()vsource.load()

Preloads the function cache with a specified validity duration.

js	function **load**(**msValidity**)
php	function **load**(**$msValidity**)
cpp	YRETCODE **load**(int **msValidity**)
m	-(YRETCODE) **load** : (int) **msValidity**
pas	function **load**(**msValidity**: integer): YRETCODE
vb	function **load**(ByVal **msValidity** As Integer) As YRETCODE
cs	YRETCODE **load**(int **msValidity**)
java	int **load**(long **msValidity**)
py	def **load**(**msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network trafic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

vsource→nextVSource()vsource.nextVSource() YVSource

Continues the enumeration of voltage sources started using yFirstVSource().

YVSource **nextVSource**( )

vsource→nextVSource()vsource.nextVSource()

Continues the enumeration of voltage sources started using yFirstVSource().

js	function **nextVSource**( )
php	function **nextVSource**( )
cpp	YVSource * **nextVSource**( )
m	-(YVSource*) **nextVSource**
pas	function **nextVSource**( ): TYVSource
vb	function **nextVSource**( ) As YVSource
cs	YVSource **nextVSource**( )
java	YVSource **nextVSource**( )
py	def **nextVSource**( )

Returns :

a pointer to a YVSource object, corresponding to a voltage source currently online, or a null pointer if there are no more voltage sources to enumerate.

vsource→pulse()vsource.pulse() YVSource

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

int **pulse**(int **voltage**, int **ms_duration**)

vsource→pulse()vsource.pulse()

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

```
js    function pulse( voltage, ms_duration)
php   function pulse( $voltage, $ms_duration)
cpp   int pulse( int voltage, int ms_duration)
m     -(int) pulse : (int) voltage : (int) ms_duration
pas   function pulse( voltage: integer, ms_duration: integer): integer
vb    function pulse( ByVal voltage As Integer,
                      ByVal ms_duration As Integer) As Integer
cs    int pulse( int voltage, int ms_duration)
java  int pulse( int voltage, int ms_duration)
py    def pulse( voltage, ms_duration)
cmd   YVSource target pulse voltage ms_duration
```

Parameters :

voltage	pulse voltage, in millivolts
ms_duration	pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→registerValueCallback() YVSource
`vsource.registerValueCallback()`

Registers the callback function that is invoked on every change of advertised value.

void **registerValueCallback**(UpdateCallback **callback**)

vsource→registerValueCallback()`vsource.registerValueCallback()`

Registers the callback function that is invoked on every change of advertised value.

`js` function **registerValueCallback**(**callback**)
`php` function **registerValueCallback**(**$callback**)
`cpp` void **registerValueCallback**(YDisplayUpdateCallback **callback**)
`pas` procedure **registerValueCallback**(**callback**: TGenericUpdateCallback)
`vb` procedure **registerValueCallback**(ByVal **callback** As GenericUpdateCallback)
`cs` void **registerValueCallback**(UpdateCallback **callback**)
`java` void **registerValueCallback**(UpdateCallback **callback**)
`py` def **registerValueCallback**(**callback**)
`m` -(void) **registerValueCallback** : (YFunctionUpdateCallback) **callback**

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

vsource→set_logicalName() YVSource
vsource→setLogicalName()
vsource.set_logicalName()

Changes the logical name of the voltage source.

int **set_logicalName**(String **newval**)

vsource→set_logicalName()
vsource→setLogicalName()vsource.set_logicalName()

Changes the logical name of the voltage source.

js	function **set_logicalName**(**newval**)
php	function **set_logicalName**(**$newval**)
cpp	int **set_logicalName**(const string& **newval**)
m	-(int) setLogicalName : (NSString*) **newval**
pas	function **set_logicalName**(**newval**: string): integer
vb	function **set_logicalName**(ByVal **newval** As String) As Integer
cs	int **set_logicalName**(string **newval**)
java	int **set_logicalName**(String **newval**)
py	def **set_logicalName**(**newval**)
cmd	YVSource **target set_logicalName newval**

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage source

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→set_userData() YVSource
vsource→setUserData()vsource.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

vsource→set_userData()
vsource→setUserData()vsource.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

js	function **set_userData**(**data**)
php	function **set_userData**(**$data**)
cpp	void **set_userData**(void* **data**)
m	-(void) setUserData : (void*) **data**
pas	procedure **set_userData**(**data**: Tobject)
vb	procedure **set_userData**(ByVal **data** As Object)
cs	void **set_userData**(object **data**)
java	void **set_userData**(Object **data**)
py	def **set_userData**(**data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

vsource→set_voltage() YVSource
vsource→setVoltage()vsource.set_voltage()

Tunes the device output voltage (milliVolts).

int **set_voltage**(int **newval**)

vsource→set_voltage()
vsource→setVoltage()vsource.set_voltage()

Tunes the device output voltage (milliVolts).

```
js    function set_voltage( newval)
php   function set_voltage( $newval)
cpp   int set_voltage( int newval)
m     -(int) setVoltage : (int) newval
pas   function set_voltage( newval: LongInt): integer
vb    function set_voltage( ByVal newval As Integer) As Integer
cs    int set_voltage( int newval)
java  int set_voltage( int newval)
py    def set_voltage( newval)
cmd   YVSource target set_voltage newval
```

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→voltageMove()`vsource.voltageMove()` YVSource

Performs a smooth move at constant speed toward a given value.

int **voltageMove**(int **target**, int **ms_duration**)

vsource→voltageMove()`vsource.voltageMove()`

Performs a smooth move at constant speed toward a given value.

```
js    function voltageMove( target, ms_duration)
php   function voltageMove( $target, $ms_duration)
cpp   int voltageMove( int target, int ms_duration)
m     -(int) voltageMove : (int) target : (int) ms_duration
pas   function voltageMove( target: integer, ms_duration: integer): integer
vb    function voltageMove( ByVal target As Integer,
                            ByVal ms_duration As Integer) As Integer
cs    int voltageMove( int target, int ms_duration)
java  int voltageMove( int target, int ms_duration)
py    def voltageMove( target, ms_duration)
cmd   YVSource target voltageMove target ms_duration
```

Parameters :

target	new output value at end of transition, in milliVolts.
ms_duration	transition duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.43. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
php     require_once('yocto_wakeupmonitor.php');
cpp     #include "yocto_wakeupmonitor.h"
m       #import "yocto_wakeupmonitor.h"
pas     uses yocto_wakeupmonitor;
vb      yocto_wakeupmonitor.vb
cs      yocto_wakeupmonitor.cs
java    import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py      from yocto_wakeupmonitor import *
```

Global functions
yFindWakeUpMonitor(**func**) Retrieves a monitor for a given identifier.
yFirstWakeUpMonitor() Starts the enumeration of monitors currently accessible.
`YWakeUpMonitor` methods
wakeupmonitor→describe() Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
wakeupmonitor→get_advertisedValue() Returns the current value of the monitor (no more than 6 characters).
wakeupmonitor→get_errorMessage() Returns the error message of the latest error with the monitor.
wakeupmonitor→get_errorType() Returns the numerical error code of the latest error with the monitor.
wakeupmonitor→get_friendlyName() Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.
wakeupmonitor→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
wakeupmonitor→get_functionId() Returns the hardware identifier of the monitor, without reference to the module.
wakeupmonitor→get_hardwareId() Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.
wakeupmonitor→get_logicalName() Returns the logical name of the monitor.
wakeupmonitor→get_module() Gets the `YModule` object for the device on which the function is located.
wakeupmonitor→get_module_async(**callback**, **context**) Gets the `YModule` object for the device on which the function is located (asynchronous version).
wakeupmonitor→get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)

wakeupmonitor→get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

wakeupmonitor→get_sleepCountdown()

Returns the delay before the next sleep period.

wakeupmonitor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupmonitor→get_wakeUpReason()

Returns the latest wake up reason.

wakeupmonitor→get_wakeUpState()

Returns the current state of the monitor

wakeupmonitor→isOnline()

Checks if the monitor is currently reachable, without raising any error.

wakeupmonitor→isOnline_async(**callback**, **context**)

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

wakeupmonitor→load(**msValidity**)

Preloads the monitor cache with a specified validity duration.

wakeupmonitor→load_async(**msValidity**, **callback**, **context**)

Preloads the monitor cache with a specified validity duration (asynchronous version).

wakeupmonitor→nextWakeUpMonitor()

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

wakeupmonitor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

wakeupmonitor→resetSleepCountDown()

Resets the sleep countdown.

wakeupmonitor→set_logicalName(**newval**)

Changes the logical name of the monitor.

wakeupmonitor→set_nextWakeUp(**newval**)

Changes the days of the week when a wake up must take place.

wakeupmonitor→set_powerDuration(**newval**)

Changes the maximal wake up time (seconds) before automatically going to sleep.

wakeupmonitor→set_sleepCountdown(**newval**)

Changes the delay before the next sleep period.

wakeupmonitor→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

wakeupmonitor→sleep(**secBeforeSleep**)

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepFor(**secUntilWakeUp**, **secBeforeSleep**)

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepUntil(**wakeUpTime**, **secBeforeSleep**)

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wakeupmonitor→wakeUp()

Forces a wake up.

YWakeUpMonitor.FindWakeUpMonitor() YWakeUpMonitor
yFindWakeUpMonitor()
YWakeUpMonitor.FindWakeUpMonitor()

Retrieves a monitor for a given identifier.

YWakeUpMonitor **FindWakeUpMonitor**(String **func**)

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() YWakeUpMonitor
yFirstWakeUpMonitor()
YWakeUpMonitor.FirstWakeUpMonitor()

Starts the enumeration of monitors currently accessible.

YWakeUpMonitor **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() YWakeUpMonitor
wakeupmonitor.describe()

Returns a short text that describes unambiguously the instance of the monitor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 monitor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

wakeupmonitor→get_advertisedValue() YWakeUpMonitor
wakeupmonitor→advertisedValue()
wakeupmonitor.get_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the monitor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wakeupmonitor→get_errorMessage() YWakeUpMonitor
wakeupmonitor→errorMessage()
wakeupmonitor.get_errorMessage()

Returns the error message of the latest error with the monitor.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the monitor object

wakeupmonitor→get_errorType() YWakeUpMonitor
wakeupmonitor→errorType()
wakeupmonitor.get_errorType()

Returns the numerical error code of the latest error with the monitor.

int **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() YWakeUpMonitor
wakeupmonitor→friendlyName()
wakeupmonitor.get_friendlyName()

Returns a global identifier of the monitor in the format MODULE_NAME.FUNCTION_NAME.

String **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() YWakeUpMonitor
wakeupmonitor→functionDescriptor()
wakeupmonitor.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupmonitor→get_functionId() YWakeUpMonitor
wakeupmonitor→functionId()
wakeupmonitor.get_functionId()

Returns the hardware identifier of the monitor, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the monitor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupmonitor→get_hardwareId() YWakeUpMonitor
wakeupmonitor→hardwareId()
wakeupmonitor.get_hardwareId()

Returns the unique hardware identifier of the monitor in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the monitor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the monitor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wakeupmonitor→get_logicalName() YWakeUpMonitor
wakeupmonitor→logicalName()
wakeupmonitor.get_logicalName()

Returns the logical name of the monitor.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the monitor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupmonitor→get_module() YWakeUpMonitor
wakeupmonitor→module()
wakeupmonitor.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

wakeupmonitor→get_nextWakeUp() YWakeUpMonitor
wakeupmonitor→nextWakeUp()
wakeupmonitor.get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)

long **get_nextWakeUp**( )

Returns :
an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns Y_NEXTWAKEUP_INVALID.

wakeupmonitor→get_powerDuration() YWakeUpMonitor
wakeupmonitor→powerDuration()
wakeupmonitor.get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

int **get_powerDuration**( )

Returns :
an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns `Y_POWERDURATION_INVALID`.

wakeupmonitor→get_sleepCountdown() YWakeUpMonitor
wakeupmonitor→sleepCountdown()
wakeupmonitor.get_sleepCountdown()

Returns the delay before the next sleep period.

int **get_sleepCountdown**( )

Returns :
an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns Y_SLEEPCOUNTDOWN_INVALID.

wakeupmonitor→get_userData() YWakeUpMonitor
wakeupmonitor→userData()
wakeupmonitor.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

wakeupmonitor→get_wakeUpReason() YWakeUpMonitor
wakeupmonitor→wakeUpReason()
wakeupmonitor.get_wakeUpReason()

Returns the latest wake up reason.

int **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() YWakeUpMonitor
wakeupmonitor→wakeUpState()
wakeupmonitor.get_wakeUpState()

Returns the current state of the monitor

int **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() YWakeUpMonitor
wakeupmonitor.isOnline()

Checks if the monitor is currently reachable, without raising any error.

```
boolean isOnline( )
```

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

Returns :

`true` if the monitor can be reached, and `false` otherwise

wakeupmonitor→load()wakeupmonitor.load() YWakeUpMonitor

Preloads the monitor cache with a specified validity duration.

int **load**(long **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→nextWakeUpMonitor() YWakeUpMonitor
`wakeupmonitor.nextWakeUpMonitor()`

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

YWakeUpMonitor **nextWakeUpMonitor**( )

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a `null` pointer if there are no more monitors to enumerate.

wakeupmonitor→registerValueCallback() YWakeUpMonitor
`wakeupmonitor.registerValueCallback()`

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupmonitor→resetSleepCountDown() YWakeUpMonitor
wakeupmonitor.resetSleepCountDown()

Resets the sleep countdown.

int **resetSleepCountDown**( )

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_logicalName() YWakeUpMonitor
wakeupmonitor→setLogicalName()
wakeupmonitor.set_logicalName()

Changes the logical name of the monitor.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the monitor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_nextWakeUp() YWakeUpMonitor
wakeupmonitor→setNextWakeUp()
wakeupmonitor.set_nextWakeUp()

Changes the days of the week when a wake up must take place.

int **set_nextWakeUp**(long **newval**)

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_powerDuration() YWakeUpMonitor
wakeupmonitor→setPowerDuration()
wakeupmonitor.set_powerDuration()

Changes the maximal wake up time (seconds) before automatically going to sleep.

int **set_powerDuration**(int **newval**)

Parameters :

newval an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_sleepCountdown() YWakeUpMonitor
wakeupmonitor→setSleepCountdown()
wakeupmonitor.set_sleepCountdown()

Changes the delay before the next sleep period.

int **set_sleepCountdown**(int **newval**)

Parameters :

newval an integer corresponding to the delay before the next sleep period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_userData() YWakeUpMonitor
wakeupmonitor→setUserData()
`wakeupmonitor.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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.

int **sleep**(int **secBeforeSleep**)

Parameters :

secBeforeSleep number of seconds before going into sleep mode,

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepFor() YWakeUpMonitor
wakeupmonitor.sleepFor()

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

int **sleepFor**(int **secUntilWakeUp**, int **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() YWakeUpMonitor
wakeupmonitor.sleepUntil()

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

int **sleepUntil**(int **wakeUpTime**, int **secBeforeSleep**)

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

wakeUpTime	wake-up datetime (UNIX format)
secBeforeSleep	number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→wakeUp() YWakeUpMonitor
`wakeupmonitor.wakeUp()`

Forces a wake up.

int **wakeUp**( )

3.44. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_wakeupschedule.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWakeUpSchedule = yoctolib.YWakeUpSchedule;
php     require_once('yocto_wakeupschedule.php');
cpp     #include "yocto_wakeupschedule.h"
m       #import "yocto_wakeupschedule.h"
pas     uses yocto_wakeupschedule;
vb      yocto_wakeupschedule.vb
cs      yocto_wakeupschedule.cs
java    import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
py      from yocto_wakeupschedule import *
```

Global functions
yFindWakeUpSchedule(**func**) Retrieves a wake up schedule for a given identifier.
yFirstWakeUpSchedule() Starts the enumeration of wake up schedules currently accessible.
`YWakeUpSchedule` methods
wakeupschedule→describe() Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
wakeupschedule→get_advertisedValue() Returns the current value of the wake up schedule (no more than 6 characters).
wakeupschedule→get_errorMessage() Returns the error message of the latest error with the wake up schedule.
wakeupschedule→get_errorType() Returns the numerical error code of the latest error with the wake up schedule.
wakeupschedule→get_friendlyName() Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.
wakeupschedule→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
wakeupschedule→get_functionId() Returns the hardware identifier of the wake up schedule, without reference to the module.
wakeupschedule→get_hardwareId() Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.
wakeupschedule→get_hours() Returns the hours scheduled for wake up.
wakeupschedule→get_logicalName() Returns the logical name of the wake up schedule.
wakeupschedule→get_minutes() Returns all the minutes of each hour that are scheduled for wake up.
wakeupschedule→get_minutesA()

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

wakeupschedule→get_minutesB()
Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

wakeupschedule→get_module()
Gets the `YModule` object for the device on which the function is located.

wakeupschedule→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

wakeupschedule→get_monthDays()
Returns the days of the month scheduled for wake up.

wakeupschedule→get_months()
Returns the months scheduled for wake up.

wakeupschedule→get_nextOccurence()
Returns the date/time (seconds) of the next wake up occurence

wakeupschedule→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupschedule→get_weekDays()
Returns the days of the week scheduled for wake up.

wakeupschedule→isOnline()
Checks if the wake up schedule is currently reachable, without raising any error.

wakeupschedule→isOnline_async(**callback**, **context**)
Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

wakeupschedule→load(**msValidity**)
Preloads the wake up schedule cache with a specified validity duration.

wakeupschedule→load_async(**msValidity**, **callback**, **context**)
Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

wakeupschedule→nextWakeUpSchedule()
Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

wakeupschedule→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

wakeupschedule→set_hours(**newval**)
Changes the hours when a wake up must take place.

wakeupschedule→set_logicalName(**newval**)
Changes the logical name of the wake up schedule.

wakeupschedule→set_minutes(**bitmap**)
Changes all the minutes where a wake up must take place.

wakeupschedule→set_minutesA(**newval**)
Changes the minutes in the 00-29 interval when a wake up must take place.

wakeupschedule→set_minutesB(**newval**)
Changes the minutes in the 30-59 interval when a wake up must take place.

wakeupschedule→set_monthDays(**newval**)
Changes the days of the month when a wake up must take place.

wakeupschedule→set_months(**newval**)
Changes the months when a wake up must take place.

wakeupschedule→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

wakeupschedule→set_weekDays(newval)

Changes the days of the week when a wake up must take place.

wakeupschedule→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWakeUpSchedule.FindWakeUpSchedule() YWakeUpSchedule
yFindWakeUpSchedule()
YWakeUpSchedule.FindWakeUpSchedule()

Retrieves a wake up schedule for a given identifier.

YWakeUpSchedule **FindWakeUpSchedule**(String **func**)

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() YWakeUpSchedule yFirstWakeUpSchedule() YWakeUpSchedule.FirstWakeUpSchedule()

Starts the enumeration of wake up schedules currently accessible.

YWakeUpSchedule **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() YWakeUpSchedule
wakeupschedule.describe()

Returns a short text that describes unambiguously the instance of the wake up schedule in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 wake up schedule (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

wakeupschedule→get_advertisedValue() YWakeUpSchedule
wakeupschedule→advertisedValue()
wakeupschedule.get_advertisedValue()

Returns the current value of the wake up schedule (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the wake up schedule (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wakeupschedule→get_errorMessage() YWakeUpSchedule
wakeupschedule→errorMessage()
wakeupschedule.get_errorMessage()

Returns the error message of the latest error with the wake up schedule.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the wake up schedule object

wakeupschedule→get_errorType() YWakeUpSchedule
wakeupschedule→errorType()
wakeupschedule.get_errorType()

Returns the numerical error code of the latest error with the wake up schedule.

int **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() YWakeUpSchedule
wakeupschedule→friendlyName()
wakeupschedule.get_friendlyName()

Returns a global identifier of the wake up schedule in the format MODULE_NAME.FUNCTION_NAME.

String **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() YWakeUpSchedule
wakeupschedule→functionDescriptor()
wakeupschedule.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupschedule→get_functionId() YWakeUpSchedule
wakeupschedule→functionId()
wakeupschedule.get_functionId()

Returns the hardware identifier of the wake up schedule, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the wake up schedule (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupschedule→get_hardwareId() YWakeUpSchedule
wakeupschedule→hardwareId()
wakeupschedule.get_hardwareId()

Returns the unique hardware identifier of the wake up schedule in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wake up schedule. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the wake up schedule (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wakeupschedule→get_hours() YWakeUpSchedule
wakeupschedule→hours()
wakeupschedule.get_hours()

Returns the hours scheduled for wake up.

int **get_hours**( )

Returns :
an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns Y_HOURS_INVALID.

wakeupschedule→get_logicalName() YWakeUpSchedule
wakeupschedule→logicalName()
wakeupschedule.get_logicalName()

Returns the logical name of the wake up schedule.

String **get_logicalName**( )

Returns :

a string corresponding to the logical name of the wake up schedule. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupschedule→get_minutes() YWakeUpSchedule
wakeupschedule→minutes()
wakeupschedule.get_minutes()

Returns all the minutes of each hour that are scheduled for wake up.

long **get_minutes**( )

wakeupschedule→get_minutesA() YWakeUpSchedule
wakeupschedule→minutesA()
wakeupschedule.get_minutesA()

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

int **get_minutesA**( )

Returns :
an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y_MINUTESA_INVALID.

wakeupschedule→get_minutesB() YWakeUpSchedule
wakeupschedule→minutesB()
wakeupschedule.get_minutesB()

Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

int **get_minutesB**( )

Returns :
an integer corresponding to the minutes in the 30-59 intervalof each hour scheduled for wake up

On failure, throws an exception or returns Y_MINUTESB_INVALID.

wakeupschedule→get_module() YWakeUpSchedule
wakeupschedule→module()
wakeupschedule.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

wakeupschedule→get_monthDays() YWakeUpSchedule
wakeupschedule→monthDays()
wakeupschedule.get_monthDays()

Returns the days of the month scheduled for wake up.

int **get_monthDays**( )

Returns :
an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns `Y_MONTHDAYS_INVALID`.

wakeupschedule→get_months() YWakeUpSchedule
wakeupschedule→months()
wakeupschedule.get_months()

Returns the months scheduled for wake up.

int **get_months**( )

Returns :
an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns `Y_MONTHS_INVALID`.

wakeupschedule→get_nextOccurence() YWakeUpSchedule
wakeupschedule→nextOccurence()
wakeupschedule.get_nextOccurence()

Returns the date/time (seconds) of the next wake up occurence

long **get_nextOccurence**( )

Returns :
an integer corresponding to the date/time (seconds) of the next wake up occurence

On failure, throws an exception or returns Y_NEXTOCCURENCE_INVALID.

wakeupschedule→get_userData() YWakeUpSchedule
wakeupschedule→userData()
wakeupschedule.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

wakeupschedule→get_weekDays() YWakeUpSchedule
wakeupschedule→weekDays()
wakeupschedule.get_weekDays()

Returns the days of the week scheduled for wake up.

int **get_weekDays**( )

Returns :
an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns `Y_WEEKDAYS_INVALID`.

wakeupschedule→isOnline() YWakeUpSchedule
wakeupschedule.isOnline()

Checks if the wake up schedule is currently reachable, without raising any error.

```
boolean isOnline( )
```

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

Returns :

`true` if the wake up schedule can be reached, and `false` otherwise

wakeupschedule→load()wakeupschedule.load() YWakeUpSchedule

Preloads the wake up schedule cache with a specified validity duration.

int **load**(long **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→nextWakeUpSchedule() YWakeUpSchedule
wakeupschedule.nextWakeUpSchedule()

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

YWakeUpSchedule **nextWakeUpSchedule**( )

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to a wake up schedule currently online, or a `null` pointer if there are no more wake up schedules to enumerate.

wakeupschedule→registerValueCallback() YWakeUpSchedule
wakeupschedule.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupschedule→set_hours() YWakeUpSchedule
wakeupschedule→setHours()
wakeupschedule.set_hours()

Changes the hours when a wake up must take place.

int **set_hours**(int **newval**)

Parameters :

newval an integer corresponding to the hours when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_logicalName() YWakeUpSchedule
wakeupschedule→setLogicalName()
wakeupschedule.set_logicalName()

Changes the logical name of the wake up schedule.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wake up schedule.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutes() YWakeUpSchedule
wakeupschedule→setMinutes()
wakeupschedule.set_minutes()

Changes all the minutes where a wake up must take place.

```
int set_minutes( long bitmap)
```

Parameters :

bitmap Minutes 00-59 of each hour scheduled for wake up.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesA() YWakeUpSchedule
wakeupschedule→setMinutesA()
wakeupschedule.set_minutesA()

Changes the minutes in the 00-29 interval when a wake up must take place.

int **set_minutesA**(int **newval**)

Parameters :

newval an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesB() YWakeUpSchedule
wakeupschedule→setMinutesB()
wakeupschedule.set_minutesB()

Changes the minutes in the 30-59 interval when a wake up must take place.

int **set_minutesB**(int **newval**)

Parameters :

newval an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_monthDays() YWakeUpSchedule
wakeupschedule→setMonthDays()
wakeupschedule.set_monthDays()

Changes the days of the month when a wake up must take place.

int **set_monthDays**(int **newval**)

Parameters :

newval an integer corresponding to the days of the month when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_months() YWakeUpSchedule
wakeupschedule→setMonths()
wakeupschedule.set_months()

Changes the months when a wake up must take place.

```
int set_months( int newval)
```

Parameters :

newval an integer corresponding to the months when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_userData() YWakeUpSchedule
wakeupschedule→setUserData()
wakeupschedule.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

wakeupschedule→set_weekDays() YWakeUpSchedule
wakeupschedule→setWeekDays()
wakeupschedule.set_weekDays()

Changes the days of the week when a wake up must take place.

int **set_weekDays**(int **newval**)

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.45. Watchdog function interface

The watchog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven direcly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_watchdog.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWatchdog = yoctolib.YWatchdog;
php     require_once('yocto_watchdog.php');
cpp     #include "yocto_watchdog.h"
m       #import "yocto_watchdog.h"
pas     uses yocto_watchdog;
vb      yocto_watchdog.vb
cs      yocto_watchdog.cs
java    import com.yoctopuce.YoctoAPI.YWatchdog;
py      from yocto_watchdog import *
```

Global functions
yFindWatchdog(**func**) Retrieves a watchdog for a given identifier.
yFirstWatchdog() Starts the enumeration of watchdog currently accessible.
`YWatchdog` methods
watchdog→delayedPulse(**ms_delay**, **ms_duration**) Schedules a pulse.
watchdog→describe() Returns a short text that describes unambiguously the instance of the watchdog in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
watchdog→get_advertisedValue() Returns the current value of the watchdog (no more than 6 characters).
watchdog→get_autoStart() Returns the watchdog runing state at module power on.
watchdog→get_countdown() Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.
watchdog→get_errorMessage() Returns the error message of the latest error with the watchdog.
watchdog→get_errorType() Returns the numerical error code of the latest error with the watchdog.
watchdog→get_friendlyName() Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.
watchdog→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
watchdog→get_functionId() Returns the hardware identifier of the watchdog, without reference to the module.

watchdog→get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

watchdog→get_logicalName()

Returns the logical name of the watchdog.

watchdog→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

watchdog→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

watchdog→get_module()

Gets the `YModule` object for the device on which the function is located.

watchdog→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

watchdog→get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

watchdog→get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

watchdog→get_running()

Returns the watchdog running state.

watchdog→get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

watchdog→get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

watchdog→get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

watchdog→get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

watchdog→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

watchdog→isOnline()

Checks if the watchdog is currently reachable, without raising any error.

watchdog→isOnline_async(callback, context)

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

watchdog→load(msValidity)

Preloads the watchdog cache with a specified validity duration.

watchdog→load_async(msValidity, callback, context)

Preloads the watchdog cache with a specified validity duration (asynchronous version).

watchdog→nextWatchdog()

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

watchdog→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

watchdog→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

watchdog→resetWatchdog()

Resets the watchdog.

watchdog→set_autoStart(**newval**)

Changes the watchdog runningsttae at module power on.

watchdog→set_logicalName(**newval**)

Changes the logical name of the watchdog.

watchdog→set_maxTimeOnStateA(**newval**)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

watchdog→set_maxTimeOnStateB(**newval**)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

watchdog→set_output(**newval**)

Changes the output state of the watchdog, when used as a simple switch (single throw).

watchdog→set_running(**newval**)

Changes the running state of the watchdog.

watchdog→set_state(**newval**)

Changes the state of the watchdog (A for the idle position, B for the active position).

watchdog→set_stateAtPowerOn(**newval**)

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

watchdog→set_triggerDelay(**newval**)

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

watchdog→set_triggerDuration(**newval**)

Changes the duration of resets caused by the watchdog, in milliseconds.

watchdog→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

watchdog→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWatchdog.FindWatchdog() YWatchdog yFindWatchdog()YWatchdog.FindWatchdog()

Retrieves a watchdog for a given identifier.

YWatchdog **FindWatchdog**(String **func**)

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() YWatchdog yFirstWatchdog()YWatchdog.FirstWatchdog()

Starts the enumeration of watchdog currently accessible.

YWatchdog **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() YWatchdog
`watchdog.delayedPulse()`

Schedules a pulse.

int **delayedPulse**(int **ms_delay**, int **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.

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 watchdog (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

watchdog→get_advertisedValue() YWatchdog
watchdog→advertisedValue()
`watchdog.get_advertisedValue()`

Returns the current value of the watchdog (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the watchdog (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

watchdog→get_autoStart() YWatchdog
watchdog→autoStart()`watchdog.get_autoStart()`

Returns the watchdog runing state at module power on.

int **get_autoStart**( )

Returns :

either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog runing state at module power on

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

watchdog→get_countdown() YWatchdog
watchdog→countdown()
watchdog.get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

long **get_countdown**( )

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

watchdog→get_errorMessage() YWatchdog
watchdog→errorMessage()
watchdog.get_errorMessage()

Returns the error message of the latest error with the watchdog.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the watchdog object

watchdog→get_errorType() YWatchdog
watchdog→errorType()
watchdog.get_errorType()

Returns the numerical error code of the latest error with the watchdog.

int **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() YWatchdog
watchdog→friendlyName()
watchdog.get_friendlyName()

Returns a global identifier of the watchdog in the format MODULE_NAME.FUNCTION_NAME.

String **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() YWatchdog
watchdog→functionDescriptor()
watchdog.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

watchdog→get_functionId() YWatchdog
watchdog→functionId()
watchdog.get_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the watchdog (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

watchdog→get_hardwareId() YWatchdog
watchdog→hardwareId()
watchdog.get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the watchdog (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

watchdog→get_logicalName() YWatchdog
watchdog→logicalName()
watchdog.get_logicalName()

Returns the logical name of the watchdog.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the watchdog. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

watchdog→get_maxTimeOnStateA() YWatchdog
watchdog→maxTimeOnStateA()
watchdog.get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

long **get_maxTimeOnStateA**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.

watchdog→get_maxTimeOnStateB() YWatchdog
watchdog→maxTimeOnStateB()
watchdog.get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

long **get_maxTimeOnStateB**( )

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEB_INVALID.

watchdog→get_module() YWatchdog
watchdog→module()watchdog.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

watchdog→get_output() YWatchdog
watchdog→output()watchdog.get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

int **get_output**( )

Returns :

either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.

watchdog→get_pulseTimer() YWatchdog
watchdog→pulseTimer()
watchdog.get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

long **get_pulseTimer**( )

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

watchdog→get_running() YWatchdog
watchdog→running()watchdog.get_running()

Returns the watchdog running state.

int **get_running**( )

Returns :

either Y_RUNNING_OFF or Y_RUNNING_ON, according to the watchdog running state

On failure, throws an exception or returns Y_RUNNING_INVALID.

watchdog→get_state() YWatchdog
watchdog→state()watchdog.get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

int **get_state**( )

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

watchdog→get_stateAtPowerOn() YWatchdog
watchdog→stateAtPowerOn()
watchdog.get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

int **get_stateAtPowerOn**( )

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

watchdog→get_triggerDelay() YWatchdog
watchdog→triggerDelay()
watchdog.get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

long **get_triggerDelay**( )

Returns :

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns Y_TRIGGERDELAY_INVALID.

watchdog→get_triggerDuration() YWatchdog
watchdog→triggerDuration()
watchdog.get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

long **get_triggerDuration**( )

Returns :
an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns Y_TRIGGERDURATION_INVALID.

watchdog→get_userData() YWatchdog
watchdog→userData()watchdog.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

Returns :
`true` if the watchdog can be reached, and `false` otherwise

watchdog→load()`watchdog.load()` YWatchdog

Preloads the watchdog cache with a specified validity duration.

int **load**(long **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→nextWatchdog() YWatchdog
watchdog.nextWatchdog()

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

YWatchdog **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).

int **pulse**(int **ms_duration**)

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→registerValueCallback() YWatchdog
watchdog.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

watchdog→resetWatchdog() YWatchdog
watchdog.resetWatchdog()

Resets the watchdog.

```
int resetWatchdog( )
```

When the watchdog is running, this function must be called on a regular basis to prevent the watchog to trigger

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_autoStart() YWatchdog
watchdog→setAutoStart()
watchdog.set_autoStart()

Changes the watchdog runningsttae at module power on.

int **set_autoStart**(int **newval**)

Remember to call the saveToFlash() method and then to reboot the module to apply this setting.

Parameters :

newval either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog runningsttae at module power on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_logicalName() YWatchdog
watchdog→setLogicalName()
watchdog.set_logicalName()

Changes the logical name of the watchdog.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the watchdog.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateA() YWatchdog
watchdog→setMaxTimeOnStateA()
watchdog.set_maxTimeOnStateA()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

int **set_maxTimeOnStateA**(long **newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateB() YWatchdog
watchdog→setMaxTimeOnStateB()
watchdog.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

int **set_maxTimeOnStateB**(long **newval**)

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_output() YWatchdog
watchdog→setOutput()watchdog.set_output()

Changes the output state of the watchdog, when used as a simple switch (single throw).

int **set_output**(int **newval**)

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_running() YWatchdog
watchdog→setRunning()watchdog.set_running()

Changes the running state of the watchdog.

int **set_running**(int **newval**)

Parameters :

newval either Y_RUNNING_OFF or Y_RUNNING_ON, according to the running state of the watchdog

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_state() YWatchdog
watchdog→setState()watchdog.set_state()

Changes the state of the watchdog (A for the idle position, B for the active position).

int **set_state**(int **newval**)

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_stateAtPowerOn() YWatchdog
watchdog→setStateAtPowerOn()
watchdog.set_stateAtPowerOn()

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

int **set_stateAtPowerOn**(int **newval**)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDelay() YWatchdog
watchdog→setTriggerDelay()
watchdog.set_triggerDelay()

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

int **set_triggerDelay**(long **newval**)

Parameters :

newval an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDuration() YWatchdog
watchdog→setTriggerDuration()
watchdog.set_triggerDuration()

Changes the duration of resets caused by the watchdog, in milliseconds.

int **set_triggerDuration**(long **newval**)

Parameters :

newval an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_userData() YWatchdog
watchdog→setUserData()
watchdog.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **data**)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.46. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_wireless.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWireless = yoctolib.YWireless;
php     require_once('yocto_wireless.php');
cpp     #include "yocto_wireless.h"
m       #import "yocto_wireless.h"
pas     uses yocto_wireless;
vb      yocto_wireless.vb
cs      yocto_wireless.cs
java    import com.yoctopuce.YoctoAPI.YWireless;
py      from yocto_wireless import *
```

Global functions

yFindWireless(**func**)

Retrieves a wireless lan interface for a given identifier.

yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

`YWireless` methods

wireless→adhocNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

wireless→describe()

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wireless→get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

wireless→get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

wireless→get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

wireless→get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

wireless→get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

wireless→get_friendlyName()

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

wireless→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

wireless→get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

wireless→get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

wireless→get_linkQuality()

Returns the link quality, expressed in percent.

wireless→get_logicalName()

Returns the logical name of the wireless lan interface.

wireless→get_message()

Returns the latest status message from the wireless interface.

wireless→get_module()

Gets the `YModule` object for the device on which the function is located.

wireless→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

wireless→get_security()

Returns the security algorithm used by the selected wireless network.

wireless→get_ssid()

Returns the wireless network name (SSID).

wireless→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

wireless→isOnline()

Checks if the wireless lan interface is currently reachable, without raising any error.

wireless→isOnline_async(**callback**, **context**)

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

wireless→joinNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

wireless→load(**msValidity**)

Preloads the wireless lan interface cache with a specified validity duration.

wireless→load_async(**msValidity**, **callback**, **context**)

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

wireless→nextWireless()

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

wireless→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

wireless→set_logicalName(**newval**)

Changes the logical name of the wireless lan interface.

wireless→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

wireless→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWireless.FindWireless() YWireless yFindWireless()YWireless.FindWireless()

Retrieves a wireless lan interface for a given identifier.

```
YWireless FindWireless( String func)
```

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() YWireless yFirstWireless()YWireless.FirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

YWireless **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() YWireless
wireless.adhocNetwork()

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

int **adhocNetwork**(String **ssid**, String **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.

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 wireless lan interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

wireless→get_advertisedValue() YWireless
wireless→advertisedValue()
`wireless.get_advertisedValue()`

Returns the current value of the wireless lan interface (no more than 6 characters).

String **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the wireless lan interface (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wireless→get_channel() YWireless
wireless→channel()`wireless.get_channel()`

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

int **get_channel**( )

Returns :
an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns `Y_CHANNEL_INVALID`.

wireless→get_detectedWlans() YWireless
wireless→detectedWlans()
wireless.get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

ArrayList<YWlanRecord> **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() YWireless
wireless→errorMessage()
wireless.get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

String **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the wireless lan interface object

wireless→get_errorType() YWireless
wireless→errorType()`wireless.get_errorType()`

Returns the numerical error code of the latest error with the wireless lan interface.

int **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.get_friendlyName()

Returns a global identifier of the wireless lan interface in the format MODULE_NAME.FUNCTION_NAME.

String **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() YWireless
wireless→functionDescriptor()
wireless.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wireless→get_functionId() YWireless
wireless→functionId()
wireless.get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the wireless lan interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wireless→get_hardwareId() YWireless
wireless→hardwareId()
wireless.get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form SERIAL.FUNCTIONID.

String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wireless lan interface. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the wireless lan interface (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wireless→get_linkQuality() YWireless
wireless→linkQuality()
wireless.get_linkQuality()

Returns the link quality, expressed in percent.

int **get_linkQuality**( )

Returns :
an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns Y_LINKQUALITY_INVALID.

wireless→get_logicalName() YWireless
wireless→logicalName()
wireless.get_logicalName()

Returns the logical name of the wireless lan interface.

String **get_logicalName**( )

Returns :
a string corresponding to the logical name of the wireless lan interface. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wireless→get_message() YWireless
wireless→message()`wireless.get_message()`

Returns the latest status message from the wireless interface.

String **get_message**( )

Returns :
a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns `Y_MESSAGE_INVALID`.

wireless→get_module() YWireless
wireless→module()wireless.get_module()

Gets the YModule object for the device on which the function is located.

YModule **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

wireless→get_security() YWireless
wireless→security()wireless.get_security()

Returns the security algorithm used by the selected wireless network.

int **get_security**( )

Returns :

a value among Y_SECURITY_UNKNOWN, Y_SECURITY_OPEN, Y_SECURITY_WEP, Y_SECURITY_WPA and Y_SECURITY_WPA2 corresponding to the security algorithm used by the selected wireless network

On failure, throws an exception or returns Y_SECURITY_INVALID.

wireless→get_ssid() YWireless
wireless→ssid()wireless.get_ssid()

Returns the wireless network name (SSID).

String **get_ssid**( )

Returns :
a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns Y_SSID_INVALID.

wireless→get_userData() YWireless
wireless→userData()`wireless.get_userData()`

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object **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.

boolean **isOnline**( )

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

Returns :

`true` if the wireless lan interface can be reached, and `false` otherwise

wireless→joinNetwork()`wireless.joinNetwork()` YWireless

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

int **joinNetwork**(String **ssid**, String **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.

int **load**(long **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→nextWireless() YWireless
wireless.nextWireless()

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

YWireless **nextWireless**( )

Returns :

a pointer to a `YWireless` object, corresponding to a wireless lan interface currently online, or a `null` pointer if there are no more wireless lan interfaces to enumerate.

wireless→registerValueCallback() YWireless
wireless.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

int **registerValueCallback**(UpdateCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wireless→set_logicalName() YWireless
wireless→setLogicalName()
wireless.set_logicalName()

Changes the logical name of the wireless lan interface.

int **set_logicalName**(String **newval**)

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wireless lan interface.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wireless→set_userData() YWireless
wireless→setUserData()`wireless.set_userData()`

Stores a user context provided as argument in the userData attribute of the function.

void **set_userData**(Object **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

Index

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y