



# Java for Android API Reference



# Table of contents

<b>1. Introduction</b>	<b>1</b>
<b>2. Using the Yocto-Demo with Android</b>	<b>3</b>
2.1. Native access and VirtualHub	3
2.2. Getting ready	3
2.3. Compatibility	3
2.4. Activating the USB port under Android	4
2.5. Control of the Led function	6
2.6. Control of the module part	8
2.7. Error handling	13
Blueprint	16
<b>3. Reference</b>	<b>16</b>
3.1. General functions	17
3.2. Accelerometer function interface	36
3.3. AnButton function interface	78
3.4. CarbonDioxide function interface	116
3.5. ColorLed function interface	155
3.6. Compass function interface	184
3.7. Current function interface	224
3.8. DataLogger function interface	263
3.9. Formatted data sequence	294
3.10. Recorded data sequence	304
3.11. Unformatted data sequence	316
3.12. Digital IO function interface	331
3.13. Display function interface	375
3.14. DisplayLayer object interface	422
3.15. External power supply control interface	454
3.16. Files function interface	479
3.17. GenericSensor function interface	506
3.18. Gyroscope function interface	552
3.19. Yocto-hub port interface	603
3.20. Humidity function interface	628
3.21. Led function interface	667
3.22. LightSensor function interface	694

3.23. Magnetometer function interface .....	734
3.24. Measured value .....	776
3.25. Module control interface .....	782
3.26. Network function interface .....	818
3.27. OS control .....	875
3.28. Power function interface .....	898
3.29. Pressure function interface .....	941
3.30. Pwm function interface .....	980
3.31. PwmPowerSource function interface .....	1018
3.32. Quaternion interface .....	1041
3.33. Real Time Clock function interface .....	1080
3.34. Reference frame configuration .....	1107
3.35. Relay function interface .....	1143
3.36. Sensor function interface .....	1179
3.37. Servo function interface .....	1218
3.38. Temperature function interface .....	1253
3.39. Tilt function interface .....	1294
3.40. Voc function interface .....	1333
3.41. Voltage function interface .....	1372
3.42. Voltage source function interface .....	1411
3.43. WakeUpMonitor function interface .....	1443
3.44. WakeUpSchedule function interface .....	1478
3.45. Watchdog function interface .....	1515
3.46. Wireless function interface .....	1560

<b>Index .....</b>	<b>1589</b>
--------------------	-------------

# 1. Introduction

This manual is intended to be used as a reference for Yoctopuce Java for Android 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 Android

To tell the truth, Android is not a programming language, it is an operating system developed by Google for mobile appliances such as smart phones and tablets. But it so happens that under Android everything is programmed with the same programming language: Java. Nevertheless, the programming paradigms and the possibilities to access the hardware are slightly different from classical Java, and this justifies a separate chapter on Android programming.

### 2.1. Native access and VirtualHub

In the opposite to the classical Java API, the Java for Android API can access USB modules natively. However, as there is no VirtualHub running under Android, it is not possible to remotely control Yoctopuce modules connected to a machine under Android. Naturally, the Java for Android API remains perfectly able to connect itself to a VirtualHub running on another OS.

### 2.2. Getting ready

Go to the Yoctopuce web site and download the Java for Android programming library<sup>1</sup>. The library is available as source files, and also as a jar file. Connect your modules, decompress the library files in the directory of your choice, and configure your Android programming environment so that it can find them.

To keep them simple, all the examples provided in this documentation are snippets of Android applications. You must integrate them in your own Android applications to make them work. However, you can find complete applications in the examples provided with the Java for Android library.

### 2.3. Compatibility

In an ideal world, you would only need to have a smart phone running under Android to be able to make Yoctopuce modules work. Unfortunately, it is not quite so in the real world. A machine running under Android must fulfil to a few requirements to be able to manage Yoctopuce USB modules natively.

---

<sup>1</sup> [www.yoctopuce.com/EN/libraries.php](http://www.yoctopuce.com/EN/libraries.php)

## Android 4.x

Android 4.0 (api 14) and following are officially supported. Theoretically, support of USB *host* functions since Android 3.1. But be aware that the Yoctopuce Java for Android API is regularly tested only from Android 4 onwards.

## USB *host* support

Naturally, not only must your machine have a USB port, this port must also be able to run in *host* mode. In *host* mode, the machine literally takes control of the devices which are connected to it. The USB ports of a desktop computer, for example, work in *host* mode. The opposite of the *host* mode is the *device* mode. USB keys, for instance, work in *device* mode: they must be controlled by a *host*. Some USB ports are able to work in both modes, they are *OTG (On The Go)* ports. It so happens that many mobile devices can only work in *device* mode: they are designed to be connected to a charger or a desktop computer, and nothing else. It is therefore highly recommended to pay careful attention to the technical specifications of a product working under Android before hoping to make Yoctopuce modules work with it.

Unfortunately, having a correct version of Android and USB ports working in *host* mode is not enough to guaranty that Yoctopuce modules will work well under Android. Indeed, some manufacturers configure their Android image so that devices other than keyboard and mass storage are ignored, and this configuration is hard to detect. As things currently stand, the best way to know if a given Android machine works with Yoctopuce modules consists in trying.

## Supported hardware

The library is tested and validated on the following machines:

- Samsung Galaxy S3
- Samsung Galaxy Note 2
- Google Nexus 5
- Google Nexus 7
- Acer Iconia Tab A200
- Asus Tranformer Pad TF300T
- Kurio 7

If your Android machine is not able to control Yoctopuce modules natively, you still have the possibility to remotely control modules driven by a VirtualHub on another OS, or a YoctoHub <sup>2</sup>.

## 2.4. Activating the USB port under Android

By default, Android does not allow an application to access the devices connected to the USB port. To enable your application to interact with a Yoctopuce module directly connected on your tablet on a USB port, a few additional steps are required. If you intend to interact only with modules connected on another machine through the network, you can ignore this section.

In your `AndroidManifest.xml`, you must declare using the "USB Host" functionality by adding the `<uses-feature android:name="android.hardware.usb.host" />` tag in the `manifest` section.

```
<manifest ...>
...
<uses-feature android:name="android.hardware.usb.host" />;
...
</manifest>
```

When first accessing a Yoctopuce module, Android opens a window to inform the user that the application is going to access the connected module. The user can deny or authorize access to the device. If the user authorizes the access, the application can access the connected device as long as

---

<sup>2</sup> Yoctohubs are a plug and play way to add network connectivity to your Yoctopuce devices. more info on <http://www.yoctopuce.com/EN/products/category/extensions-and-networking>



it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, you must provide a pointer on the application context by calling the `EnableUSBHost` method of the `YAPI` class before the first USB access. This function takes as arguments an object of the `android.content.Context` class (or of a subclass). As the `Activity` class is a subclass of `Context`, it is simpler to call `YAPI.EnableUSBHost(this)` ; in the method `onCreate` of your application. If the object passed as parameter is not of the correct type, a `YAPI_Exception` exception is generated.

```
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
        // Pass the application Context to the Yoctopuce Library
        YAPI.EnableUSBHost(this);
    } catch (YAPI_Exception e) {
        Log.e("Yocto", e.getLocalizedMessage());
    }
}
...
```

## Autorun

It is possible to register your application as a default application for a USB module. In this case, as soon as a module is connected to the system, the application is automatically launched. You must add `<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"/>` in the section `<intent-filter>` of the main activity. The section `<activity>` must have a pointer to an XML file containing the list of USB modules which can run the application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
...
<uses-feature android:name="android.hardware.usb.host" />
...
<application ... >
    <activity
        android:name=".MainActivity" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

        <meta-data
            android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
            android:resource="@xml/device_filter" />
        </activity>
    </application>
</manifest>
```

The XML file containing the list of modules allowed to run the application must be saved in the `res/xml` directory. This file contains a list of USB *vendorId* and *deviceId* in decimal. The following example runs the application as soon as a Yocto-Relay or a YoctoPowerRelay is connected. You can find the *vendorId* and the *deviceId* of Yoctopuce modules in the characteristics section of the documentation.

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="9440" product-id="12" />
    <usb-device vendor-id="9440" product-id="13" />
</resources>
```

## 2.5. 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.

```
[...]

// Retrieving the object representing the module (connected here locally by USB)
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
led = YLed.FindLed("YCTOPOC1-123456.led");

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

[...]
```

Let us look at these lines in more details.

### YAPI.EnableUSBHost

The `YAPI.EnableUSBHost` function initializes the API with the Context of the current application. This function takes as argument an object of the `android.content.Context` class (or of a subclass). If you intend to connect your application only to other machines through the network, this function is facultative.

### 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 string "usb" is passed as parameter, the API works with modules locally connected to the machine. 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 your Java environment and open the corresponding sample project provided in the directory **Examples//Doc-Examples** of the Yoctopuce library.

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YLed;

public class GettingStarted_Yocto_Demo extends Activity implements OnItemClickListener
{

    private YLed led = null;
    private ArrayAdapter<String> aa;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_demo);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemClickListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart()
    {
        super.onStart();

        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YLed r = YLed.FirstLed();
            while (r != null) {
                String hwid = r.get_hardwareId();
                aa.add(hwid);
                r = r.nextLed();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        // refresh Spinner with detected relay
        aa.notifyDataSetChanged();
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }

    @Override
    public void onItemClick(AdapterView<?> parent, View view, int pos, long id)
    {
        String hwid = parent.getItemAtPosition(pos).toString();
        led = YLed.FindLed(hwid);
    }
}
```

```

@Override
public void onNothingSelected(AdapterView<?> arg0)
{
}

/** Called when the user touches the button State A */
public void setLedOn(View view)
{
    // Do something in response to button click
    if (led != null)
        try {
            led.setPower(YLed.POWER_ON);
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
}

/** Called when the user touches the button State B */
public void setLedOff(View view)
{
    // Do something in response to button click
    if (led != null)
        try {
            led.setPower(YLed.POWER_OFF);
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
}
}

```

## 2.6. 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.

```

package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class ModuleControl extends Activity implements OnItemClickListener
{
    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.modulecontrol);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemClickListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart()

```

```

{
    super.onStart();

    try {
        aa.clear();
        YAPI.EnableUSBHost(this);
        YAPI.RegisterHub("usb");
        YModule r = YModule.FirstModule();
        while (r != null) {
            String hwid = r.get_hardwareId();
            aa.add(hwid);
            r = r.nextModule();
        }
    } catch (YAPI_Exception e) {
        e.printStackTrace();
    }
    // refresh Spinner with detected relay
    aa.notifyDataSetChanged();
}

@Override
protected void onStop()
{
    super.onStop();
    YAPI.FreeAPI();
}

private void DisplayModuleInfo()
{
    TextView field;
    if (module == null)
        return;
    try {
        field = (TextView) findViewById(R.id.serialfield);
        field.setText(module.getSerialNumber());
        field = (TextView) findViewById(R.id.logicalnamefield);
        field.setText(module.getLogicalName());
        field = (TextView) findViewById(R.id.luminosityfield);
        field.setText(String.format("%d%", module.getLuminosity()));
        field = (TextView) findViewById(R.id.uptimefield);
        field.setText(module.getUpTime() / 1000 + " sec");
        field = (TextView) findViewById(R.id.usbcurrentfield);
        field.setText(module.getUsbCurrent() + " mA");
        Switch sw = (Switch) findViewById(R.id.beaconswitch);
        Log.d("switch", "beacon" + module.get_beacon());
        sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
        field = (TextView) findViewById(R.id.logs);
        field.setText(module.get_lastLogs());

    } catch (YAPI_Exception e) {
        e.printStackTrace();
    }
}

@Override
public void onItemClick(AdapterView<?> parent, View view, int pos, long id)
{
    String hwid = parent.getItemAtPosition(pos).toString();
    module = YModule.FindModule(hwid);
    DisplayModuleInfo();
}

@Override
public void onNothingSelected(AdapterView<?> arg0)
{
}

public void refreshInfo(View view)
{
    DisplayModuleInfo();
}

public void toggleBeacon(View view)
{
    if (module == null)
        return;
    boolean on = ((Switch) view).isChecked();

```

```

        try {
            if (on) {
                module.setBeacon(YModule.BEACON_ON);
            } else {
                module.setBeacon(YModule.BEACON_OFF);
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }
}

```

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.

```

package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class SaveSettings extends Activity implements OnItemClickListener
{
    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.savesettings);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemClickListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart()
    {
        super.onStart();

        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule r = YModule.FirstModule();
            while (r != null) {
                String hwid = r.get_hardwareId();
                aa.add(hwid);
            }
        }
    }
}

```

```

        r = r.nextModule();
    }
} catch (YAPI_Exception e) {
    e.printStackTrace();
}
// refresh Spinner with detected relay
aa.notifyDataSetChanged();
}

@Override
protected void onStop()
{
    super.onStop();
    YAPI.FreeAPI();
}

private void DisplayModuleInfo()
{
    TextView field;
    if (module == null)
        return;
    try {
        YAPI.UpdateDeviceList(); // fixme
        field = (TextView) findViewById(R.id.logicalnamefield);
        field.setText(module.getLogLogicalName());
    } catch (YAPI_Exception e) {
        e.printStackTrace();
    }
}

@Override
public void onItemClick(AdapterView<?> parent, View view, int pos, long id)
{
    String hwid = parent.getItemAtPosition(pos).toString();
    module = YModule.FindModule(hwid);
    DisplayModuleInfo();
}

@Override
public void onNothingSelected(AdapterView<?> arg0)
{
}

public void saveName(View view)
{
    if (module == null)
        return;

    EditText edit = (EditText) findViewById(R.id.newname);
    String newname = edit.getText().toString();
    try {
        if (!YAPI.CheckLogicalName(newname)) {
            Toast.makeText(getApplicationContext(), "Invalid name (" + newname + ")",
                Toast.LENGTH_LONG).show();
            return;
        }
        module.set_logicalName(newname);
        module.saveToFlash(); // do not forget this
        edit.setText("");
    } catch (YAPI_Exception ex) {
        ex.printStackTrace();
    }
    DisplayModuleInfo();
}
}

```

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.

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.LinearLayout;
import android.widget.TextView;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class Inventory extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.inventory);
    }

    public void refreshInventory(View view)
    {
        LinearLayout layout = (LinearLayout) findViewById(R.id.inventoryList);
        layout.removeAllViews();

        try {
            YAPI.UpdateDeviceList();
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() +
                ")";

                TextView tx = new TextView(this);
                tx.setText(line);
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onStart()
    {
        super.onStart();
        try {
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        refreshInventory(null);
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }
}
```



## 2.7. 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 for Android, 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 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');
c++	#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)**

### 3. Reference

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 as 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 your local computer, use the IP address 127.0.0.1.

**callback:** that keyword makes the API run in "*HTTP Callback*" mode. This is 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()**  
**yRegisterLogFunction()**  
**YAPI.RegisterLogFunction( )**

**YAPI**

---

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()**  
**yTriggerHubDiscovery()**  
**YAPI.TriggerHubDiscovery( )**

**YAPI**

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

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');
c++	#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.

### 3. Reference

**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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.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 occurred 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 occurred while using the accelerometer object

**accelerometer→get\_friendlyName()**

**YAccelerometer**

**accelerometer→friendlyName()**

**accelerometer.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

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

---

**accelerometer→get\_functionDescriptor()****YAccelerometer****accelerometer→functionDescriptor()****accelerometer.get\_functionDescriptor()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**accelerometer→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()**  
**accelerometer**→**setLowestValue()**  
**accelerometer.set\_lowestValue()**

---

**YAccelerometer**

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');
c++	#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)**

### 3. Reference

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` is 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(BadCustomName.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 occurred 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 occurred while using the analog input object

---

**anbutton**→**get\_friendlyName()**  
**anbutton**→**friendlyName()**  
**anbutton.get\_friendlyName()**

---

**YAnButton**

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

**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\_LASTTIMERELASED\_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()**  
**anbutton**→**pulseTimer()**  
**anbutton.get\_pulseTimer()**

---

**YAnButton**

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.getRawValue()**

---

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**anbutton**→**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');
c++	#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.

### 3. Reference

---

**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()**  
**carbondioxide.calibrateFromPoints()**

---

**YCarbonDioxide**

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**carbondioxide→describe()****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` is 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(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

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

---

**carbondioxide**→**get\_advertisedValue()**  
**carbondioxide**→**advertisedValue()**  
**carbondioxide.get\_advertisedValue()**

---

**YCarbonDioxide**

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

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 occurred 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 occurred while using the CO2 sensor object

**carbondioxide→get\_friendlyName()**

**YCarbonDioxide**

**carbondioxide→friendlyName()**

**carbondioxide.get\_friendlyName()**

---

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

**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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**carbondioxide→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()**  
**carbondioxide**→**setHighestValue()**  
**carbondioxide.set\_highestValue()**

---

**YCarbonDioxide**

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()`  
`carbondioxide→setLowestValue()`  
`carbondioxide.set_lowestValue()`

**YCarbonDioxide**

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 from RGB to HSL automatically. It is then self-evident to turn on a led with a given hue and to progressively vary its saturation or lightness. If needed, you can find more information on the difference between RGB and HSL in the section following this one.

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

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` is 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(BadCustomName.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 occurred 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 occurred while using the RGB led object

---

**colorled**→**get\_friendlyName()**  
**colorled**→**friendlyName()**  
**colorled.get\_friendlyName()**

---

**YColorLed**

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

**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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**colorled**→**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: 0xHHSSL.

**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');
c++	#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.

### 3. Reference

---

**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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.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 occurred 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 occurred while using the compass object

**compass**→**get\_friendlyName()****YCompass****compass**→**friendlyName()****compass.get\_friendlyName()**

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

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()**  
**compass**→**hardwareId()**  
**compass.get\_hardwareId()**

---

**YCompass**

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 traffic for instance.

**Parameters :**

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

**Returns :**

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



**compass→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()**  
**compass**→**setHighestValue()**  
**compass.set\_highestValue()**

---

**YCompass**

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()**  
**compass**→**setLowestValue()**  
**compass.set\_lowestValue()**

**YCompass**

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()**  
**compass**→**setReportFrequency()**  
**compass.set\_reportFrequency( )**

---

**YCompass**

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');
c++	#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.

### 3. Reference

---

**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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**current→describe()**`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` is 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(BadCustomName.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()**  
**current**→**currentValue()**  
**current.get\_currentValue()**

---

**YCurrent**

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 occurred 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 occurred while using the current sensor object

**current**→**get\_friendlyName()**

**YCurrent**

**current**→**friendlyName()**

**current.get\_friendlyName()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**current→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()**  
**current**→**setHighestValue()**  
**current.set\_highestValue()**

---

**YCurrent**

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()**  
**current**→**setLowestValue()**  
**current.set\_lowestValue()**

---

**YCurrent**

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()**  
**current**→**setResolution()**  
**current.set\_resolution()**

---

**YCurrent**

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');
c++	#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 <code>YFUN_DESCR</code> corresponding to the function.
<b><code>datalogger→get_functionId()</code></b>	Returns the hardware identifier of the data logger, without reference to the module.
<b><code>datalogger→get_hardwareId()</code></b>	Returns the unique hardware identifier of the data logger in the form <code>SERIAL . FUNCTIONID</code> .
<b><code>datalogger→get_logicalName()</code></b>	Returns the logical name of the data logger.
<b><code>datalogger→get_module()</code></b>	Gets the <code>YModule</code> object for the device on which the function is located.
<b><code>datalogger→get_module_async(callback, context)</code></b>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<b><code>datalogger→get_recording()</code></b>	Returns the current activation state of the data logger.
<b><code>datalogger→get_timeUTC()</code></b>	Returns the Unix timestamp for current UTC time, if known.
<b><code>datalogger→get_userData()</code></b>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b><code>datalogger→isOnline()</code></b>	Checks if the data logger is currently reachable, without raising any error.
<b><code>datalogger→isOnline_async(callback, context)</code></b>	Checks if the data logger is currently reachable, without raising any error (asynchronous version).
<b><code>datalogger→load(msValidity)</code></b>	Preloads the data logger cache with a specified validity duration.
<b><code>datalogger→load_async(msValidity, callback, context)</code></b>	Preloads the data logger cache with a specified validity duration (asynchronous version).
<b><code>datalogger→nextDataLogger()</code></b>	Continues the enumeration of data loggers started using <code>yFirstDataLogger()</code> .
<b><code>datalogger→registerValueCallback(callback)</code></b>	Registers the callback function that is invoked on every change of advertised value.
<b><code>datalogger→set_autoStart(newval)</code></b>	Changes the default activation state of the data logger on power up.
<b><code>datalogger→set_logicalName(newval)</code></b>	Changes the logical name of the data logger.
<b><code>datalogger→set_recording(newval)</code></b>	Changes the activation state of the data logger to start/stop recording data.
<b><code>datalogger→set_timeUTC(newval)</code></b>	Changes the current UTC time reference used for recorded data.
<b><code>datalogger→set_userData(data)</code></b>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<b><code>datalogger→wait_async(callback, context)</code></b>	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` is 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(BadCustomName.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 occurred while using the data logger object

---

**`datalogger→get_errorType()`**  
**`datalogger→errorType()`**  
**`datalogger.get_errorType( )`**

---

**YDataLogger**

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

**`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 occurred while using the data logger object

**datalogger**→**get\_friendlyName()****YDataLogger****datalogger**→**friendlyName()****datalogger.get\_friendlyName()**

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**dataLogger→nextDataLogger()**  
**dataLogger.nextDataLogger( )**

---

**YDataLogger**

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()**  
**datalogger**→**setRecording()**  
**datalogger.set\_recording()**

**YDataLogger**

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');
c++	#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
<b>datarun→get_averageValue(measureName, pos)</b> Returns the average value of the measure observed at the specified time period.
<b>datarun→get_duration()</b> Returns the duration (in seconds) of the data run.
<b>datarun→get_maxValue(measureName, pos)</b> Returns the maximal value of the measure observed at the specified time period.
<b>datarun→get_measureNames()</b> Returns the names of the measures recorded by the data logger.
<b>datarun→get_minValue(measureName, pos)</b> Returns the minimal value of the measure observed at the specified time period.
<b>datarun→get_startTimeUTC()</b> Returns the start time of the data run, relative to the Jan 1, 1970.
<b>datarun→get_valueCount()</b> Returns the number of values accessible in this run, given the selected data samples interval.
<b>datarun→get_valueInterval()</b> Returns the number of seconds covered by each value in this run.
<b>datarun→set_valueInterval(valueInterval)</b> 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 <b>get_averageValue</b> ( <b>measureName</b> , <b>pos</b> )
nodejs	function <b>get_averageValue</b> ( <b>measureName</b> , <b>pos</b> )
php	function <b>get_averageValue</b> ( <b>\$measureName</b> , <b>\$pos</b> )
java	double <b>get_averageValue</b> ( String <b>measureName</b> , int <b>pos</b> )
py	def <b>get_averageValue</b> ( <b>measureName</b> , <b>pos</b> )

**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.getMinValue()`


---

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.getMinValue()`


---

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

js	function <b>get_minValue</b> ( <b>measureName</b> , <b>pos</b> )
nodejs	function <b>get_minValue</b> ( <b>measureName</b> , <b>pos</b> )
php	function <b>get_minValue</b> ( <b>\$measureName</b> , <b>\$pos</b> )
java	double <b>get_minValue</b> ( String <b>measureName</b> , int <b>pos</b> )
py	def <b>get_minValue</b> ( <b>measureName</b> , <b>pos</b> )

**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()**  
**datarun→setValueInterval()**  
**datarun.set\_valueInterval()**

YDataRun

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 instantiated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

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

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

### 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 be 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( )**

It 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 `getColumnNames()`.

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.getRowCount()**

---

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');
c++	#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` is 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(BadCustomName.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 occurred 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 occurred while using the digital IO port object

**digitalio**→**get\_friendlyName()****YDigitalIO****digitalio**→**friendlyName()****digitalio.get\_friendlyName()**

Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

**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()**  
**digitalio**→**logicalName()**  
**digitalio.get\_logicalName()**

---

**YDigitalIO**

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**digitalio**→**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()**  
**digitalio**→**setLogicalName()**  
**digitalio.set\_logicalName()**

**YDigitalIO**

---

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');
c++	#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` is 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(BadCustomName.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()**  
**display**→**displayHeight()**  
**display.get\_displayHeight()**

---

**YDisplay**

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 occurred 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 occurred while using the display object

**display**→**get\_friendlyName()**

**YDisplay**

**display**→**friendlyName()**

**display.get\_friendlyName()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**display**→**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 one 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');
c++	#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()**  
**displaylayer.drawDisc()****YDisplayLayer**

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()**  
**displaylayer→display()**  
**displaylayer.get\_display()**

---

**YDisplayLayer**

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()**  
**displaylayer.selectEraser( )**

---

**YDisplayLayer**

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 highest 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');
c++	#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` is 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(BadCustomName.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 occurred while using the power control object



---

**dualpower→get\_errorType()**  
**dualpower→errorType()**  
**dualpower.get\_errorType( )**

---

**YDualPower**

Returns the numerical error code of the latest error with the power control.

**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 occurred while using the power control object

**dualpower→get\_extVoltage()**

**YDualPower**

**dualpower→extVoltage()**

**dualpower.get\_extVoltage()**

---

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

**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 example: `MyCustomName.relay1`)

**Returns :**

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

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

**dualpower→get\_functionDescriptor()**

**YDualPower**

**dualpower→functionDescriptor()**

**dualpower.get\_functionDescriptor()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**dualpower→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 yocotolib = require('yocotolib'); var YFiles = yocotolib.YFiles;
php	require_once('yocto_files.php');
c++	#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` is 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(BadCustomName.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()**  
**files**→**advertisedValue()**  
**files.get\_advertisedValue()**

---

**YFiles**

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.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 occurred 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 occurred while using the filesystem object

**files**→**get\_filesCount()**

**YFiles**

**files**→**filesCount()****files.get\_filesCount()**

---

Returns the number of files currently loaded in the filesystem.

**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()****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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**files**→**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');
c++	#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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

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

---

**genericsensor**→**get\_advertisedValue()**  
**genericsensor**→**advertisedValue()**  
**genericsensor.get\_advertisedValue()**

---

**YGenericSensor**

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

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 occurred 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 occurred while using the generic sensor object

**genericsensor→get\_friendlyName()**

**YGenericSensor**

**genericsensor→friendlyName()**

**genericsensor.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

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

---

**genericsensor**→**get\_functionDescriptor()****YGenericSensor****genericsensor**→**functionDescriptor()****genericsensor.get\_functionDescriptor()**

---

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

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.getUserData()**

---

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**genericsensor→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()**  
**genericsensor**→**setHighestValue()**  
**genericsensor.set\_highestValue()**

---

**YGenericSensor**

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');
c++	#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 z 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()**

### 3. Reference

	Returns the angular velocity around the Y axis of the device, as a floating point number.
<b>gyro→get_zValue()</b>	Returns the angular velocity around the Z axis of the device, as a floating point number.
<b>gyro→isOnline()</b>	Checks if the gyroscope is currently reachable, without raising any error.
<b>gyro→isOnline_async(callback, context)</b>	Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).
<b>gyro→load(msValidity)</b>	Preloads the gyroscope cache with a specified validity duration.
<b>gyro→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code> .
<b>gyro→load_async(msValidity, callback, context)</b>	Preloads the gyroscope cache with a specified validity duration (asynchronous version).
<b>gyro→nextGyro()</b>	Continues the enumeration of gyroscopes started using <code>yFirstGyro()</code> .
<b>gyro→registerAnglesCallback(callback)</b>	Registers a callback function that will be invoked each time that the estimated device orientation has changed.
<b>gyro→registerQuaternionCallback(callback)</b>	Registers a callback function that will be invoked each time that the estimated device orientation has changed.
<b>gyro→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>gyro→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>gyro→set_highestValue(newval)</b>	Changes the recorded maximal value observed.
<b>gyro→set_logFrequency(newval)</b>	Changes the datalogger recording frequency for this function.
<b>gyro→set_logicalName(newval)</b>	Changes the logical name of the gyroscope.
<b>gyro→set_lowestValue(newval)</b>	Changes the recorded minimal value observed.
<b>gyro→set_reportFrequency(newval)</b>	Changes the timed value notification frequency for this function.
<b>gyro→set_resolution(newval)</b>	Changes the resolution of the measured physical values.
<b>gyro→set_userData(data)</b>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<b>gyro→wait_async(callback, context)</b>	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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

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

---

**gyro**→**get\_advertisedValue()**  
**gyro**→**advertisedValue()**  
**gyro.get\_advertisedValue()**

---

**YGyro**

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

**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.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 occurred 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 occurred while using the gyroscope object

**gyro→get\_friendlyName()**

**YGyro**

**gyro→friendlyName()**`gyro.get_friendlyName()`

---

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

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()**  
**gyro**→**functionDescriptor()**  
**gyro.get\_functionDescriptor()**

---

**YGyro**

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()**  
**gyro→reportFrequency()**  
**gyro.get\_reportFrequency( )**

**YGyro**

---

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**gyro→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()**  
**gyro**→**setHighestValue()**  
**gyro.set\_highestValue()**

**YGyro**

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');
c++	#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` is 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(BadCustomName.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.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 occurred 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 occurred while using the Yocto-hub port object



---

**hubport**→**get\_friendlyName()****YHubPort****hubport**→**friendlyName()****hubport.get\_friendlyName()**

---

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

**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 example: `MyCustomName.relay1`)

**Returns :**

a string that uniquely identifies the Yocto-hub port using logical names (ex: `MyCustomName.relay1`)

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

**hubport**→**get\_functionDescriptor()**

**YHubPort**

**hubport**→**functionDescriptor()**

**hubport.get\_functionDescriptor()**

---

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

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()**  
**hubport**→**logicalName()**  
**hubport.get\_logicalName()**

---

**YHubPort**

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**hubport**→**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.setEnabled()`

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');
c++	#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.

### 3. Reference

---

#### **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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**humidity→describe()**`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` is 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(BadCustomName.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 occurred 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 occurred while using the humidity sensor object

**humidity→get\_friendlyName()**

**YHumidity**

**humidity→friendlyName()**

**humidity.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

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

---

**humidity→get\_functionDescriptor()****YHumidity****humidity→functionDescriptor()****humidity.get\_functionDescriptor()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**humidity→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()**  
**humidity→setReportFrequency()**  
**humidity.set\_reportFrequency( )**

**YHumidity**

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()**  
**humidity→setResolution()**  
**humidity.set\_resolution()**

**YHumidity**

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');
c++	#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()

### 3. Reference

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` is 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(BadCustomName.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.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 occurred 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 occurred while using the led object

**led**→**get\_friendlyName()**

**YLed**

**led**→**friendlyName()**`led.get_friendlyName()`

---

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

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**led**→**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');
c++	#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 ambient 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 ambient 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 ambient 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 ambient light.

**lightsensor→get\_userData()**

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

**lightsensor→isOnline()**

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

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

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

**lightsensor→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 ambient light.

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

### 3. Reference

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 ambient 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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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



**lightsensor→describe()**`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` is 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(BadCustomName.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 ambient light.

**double** **get\_currentValue()**

**Returns :**

a floating point number corresponding to the current measure for the ambient 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 occurred 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 occurred while using the light sensor object

**lightsensor→get\_friendlyName()****YLightSensor****lightsensor→friendlyName()****lightsensor.get\_friendlyName()**

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

**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 ambient light.

double **get\_highestValue()**

**Returns :**

a floating point number corresponding to the maximal value observed for the ambient 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 ambient light.

**double** **get\_lowestValue()**

**Returns :**

a floating point number corresponding to the minimal value observed for the ambient 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 ambient light.

String **get\_unit()**

**Returns :**

a string corresponding to the measuring unit for the ambient light

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

**lightsensor→get\_userdata()**

**YLightSensor**

**lightsensor→userData()**

**lightsensor.getUserData()**

---

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**lightsensor→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 ambient light.

```
int set_highestValue( double newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the ambient 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 ambient light.

```
int set_lowestValue( double newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the ambient 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()**  
**lightsensor→setUserData()**  
**lightsensor.set\_userdata()**

**YLightSensor**

---

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	<code>&lt;script type='text/javascript' src='yocto_magnetometer.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YMagnetometer = yoctolib.YMagnetometer;</code>
php	<code>require_once('yocto_magnetometer.php');</code>
c++	<code>#include "yocto_magnetometer.h"</code>
m	<code>#import "yocto_magnetometer.h"</code>
pas	<code>uses yocto_magnetometer;</code>
vb	<code>yocto_magnetometer.vb</code>
cs	<code>yocto_magnetometer.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YMagnetometer;</code>
py	<code>from yocto_magnetometer import *</code>

### 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.

### 3. Reference

#### **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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.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 occurred 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 occurred while using the magnetometer object

**magnetometer→get\_friendlyName()**

**YMagnetometer**

**magnetometer→friendlyName()**

**magnetometer.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

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

---

**magnetometer→get\_functionDescriptor()****YMagnetometer****magnetometer→functionDescriptor()****magnetometer.get\_functionDescriptor()**

---

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

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()**  
**magnetometer→logicalName()**  
**magnetometer.get\_logicalName()**

---

**YMagnetometer**

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()**  
**magnetometer→module()**  
**magnetometer.get\_module()**

---

**YMagnetometer**

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()**  
**magnetometer→resolution()**  
**magnetometer.get\_resolution()**

---

**YMagnetometer**

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()**  
**magnetometer→xValue()**  
**magnetometer.get\_xValue()**

---

**YMagnetometer**

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 traffic for instance.

**Parameters :**

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

**Returns :**

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



**magnetometer→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()**  
**magnetometer**→**setLowestValue()**  
**magnetometer.set\_lowestValue()**

---

**YMagnetometer**

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 than 1 sample per second, the timestamp may have a fractional part.

**Returns :**

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

---

**measure**→**get\_maxValue()****YMeasure****measure**→**maxValue()****measure.getMaxValue()**

---

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()**  
**measure**→**startTimeUTC()**  
**measure.get\_startTimeUTC()**

---

**YMeasure**

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 than 1 sample per second, the timestamp may have a fractional part.

**Returns :**

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the 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');
c++	#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
<b>yFindModule(func)</b> Allows you to find a module from its serial number or from its logical name.
<b>yFirstModule()</b> Starts the enumeration of modules currently accessible.
YModule methods
<b>module→describe()</b> Returns a descriptive text that identifies the module.
<b>module→download(pathname)</b> Downloads the specified built-in file and returns a binary buffer with its content.
<b>module→functionCount()</b> Returns the number of functions (beside the "module" interface) available on the module.
<b>module→functionId(functionIndex)</b> Retrieves the hardware identifier of the <i>n</i> th function on the module.
<b>module→functionName(functionIndex)</b> Retrieves the logical name of the <i>n</i> th function on the module.
<b>module→functionValue(functionIndex)</b> Retrieves the advertised value of the <i>n</i> th function on the module.
<b>module→get_beacon()</b> Returns the state of the localization beacon.
<b>module→get_errorMessage()</b> Returns the error message of the latest error with this module object.
<b>module→get_errorType()</b> Returns the numerical error code of the latest error with this module object.
<b>module→get_firmwareRelease()</b> Returns the version of the firmware embedded in the module.
<b>module→get_hardwareId()</b> Returns the unique hardware identifier of the module.
<b>module→get_icon2d()</b>

	Returns the icon of the module.
<b>module</b> → <b>get_lastLogs()</b>	Returns a string with last logs of the module.
<b>module</b> → <b>get_logicalName()</b>	Returns the logical name of the module.
<b>module</b> → <b>get_luminosity()</b>	Returns the luminosity of the module informative leds (from 0 to 100).
<b>module</b> → <b>get_persistentSettings()</b>	Returns the current state of persistent module settings.
<b>module</b> → <b>get_productId()</b>	Returns the USB device identifier of the module.
<b>module</b> → <b>get_productName()</b>	Returns the commercial name of the module, as set by the factory.
<b>module</b> → <b>get_productRelease()</b>	Returns the hardware release version of the module.
<b>module</b> → <b>get_rebootCountdown()</b>	Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
<b>module</b> → <b>get_serialNumber()</b>	Returns the serial number of the module, as set by the factory.
<b>module</b> → <b>get_upTime()</b>	Returns the number of milliseconds spent since the module was powered on.
<b>module</b> → <b>get_usbBandwidth()</b>	Returns the number of USB interfaces used by the module.
<b>module</b> → <b>get_usbCurrent()</b>	Returns the current consumed by the module on the USB bus, in milli-amps.
<b>module</b> → <b>get_userData()</b>	Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>module</b> → <b>isOnline()</b>	Checks if the module is currently reachable, without raising any error.
<b>module</b> → <b>isOnline_async(callback, context)</b>	Checks if the module is currently reachable, without raising any error.
<b>module</b> → <b>load(msValidity)</b>	Preloads the module cache with a specified validity duration.
<b>module</b> → <b>load_async(msValidity, callback, context)</b>	Preloads the module cache with a specified validity duration (asynchronous version).
<b>module</b> → <b>nextModule()</b>	Continues the module enumeration started using <code>yFirstModule()</code> .
<b>module</b> → <b>reboot(secBeforeReboot)</b>	Schedules a simple module reboot after the given number of seconds.
<b>module</b> → <b>registerLogCallback(callback)</b>	todo
<b>module</b> → <b>revertFromFlash()</b>	Reloads the settings stored in the nonvolatile memory, as when the module is powered on.
<b>module</b> → <b>saveToFlash()</b>	Saves current settings in the nonvolatile memory of the module.

### 3. Reference

---

**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.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 occurred 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 occurred while using this module object

---

**module**→**get\_firmwareRelease()****YModule****module**→**firmwareRelease()****module.get\_firmwareRelease()**

---

Returns the version of the firmware embedded in the module.

**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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**module**→**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');
c++	#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` is 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(BadCustomName.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.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 occurred 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 occurred while using the network interface object



**network**→**get\_friendlyName()****YNetwork****network**→**friendlyName()****network.get\_friendlyName()**

---

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

**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()**  
**network**→**macAddress()**  
**network.get\_macAddress()**

---

**YNetwork**

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.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()**  
**network**→**wwwWatchdogDelay()**  
**network.get\_wwwWatchdogDelay()**

---

**YNetwork**

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

**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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**network**→**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 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()**  
**network**→**setLogicalName()**  
**network.set\_logicalName()**

---

**YNetwork**

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 :**

<b>fallbackIpAddr</b>	fallback IP address, to be used when no DHCP reply is received
<b>fallbackSubnetMaskLen</b>	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
<b>fallbackRouter</b>	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');
c++	#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` is 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(BadCustomName.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.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 occurred 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 occurred while using the OS control object

---

**oscontrol**→**get\_friendlyName()**  
**oscontrol**→**friendlyName()**  
**oscontrol.get\_friendlyName()**

---

**YOsControl**

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

**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 example: `MyCustomName.relay1`)

**Returns :**

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

**oscontrol→get\_functionDescriptor()**

**YOsControl**

**oscontrol→functionDescriptor()**

**oscontrol.get\_functionDescriptor()**

---

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

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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



---

**oscontrol→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()**  
**oscontrol→setLogicalName()**  
**oscontrol.set\_logicalName()**

---

**YOsControl**

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');
c++	#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)**

### 3. Reference

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

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

---

**power**→**get\_advertisedValue()**  
**power**→**advertisedValue()**  
**power.get\_advertisedValue()**

---

**YPower**

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

**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 occurred 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 occurred while using the electrical power sensor object

---

**power**→**get\_friendlyName()****YPower****power**→**friendlyName()****power.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

**power**→**get\_functionDescriptor()**

**YPower**

**power**→**functionDescriptor()**

**power.get\_functionDescriptor()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**power→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');
c++	#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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**pressure→describe()****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` is 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(BadCustomName.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 occurred 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 occurred while using the pressure sensor object

---

**pressure**→**get\_friendlyName()**  
**pressure**→**friendlyName()**  
**pressure.get\_friendlyName()**

---

**YPressure**

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

**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 example: `MyCustomName.relay1`)

**Returns :**

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

**pressure**→**get\_functionDescriptor()**

**YPressure**

**pressure**→**functionDescriptor()**

**pressure.get\_functionDescriptor()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**pressure→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');
c++	#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.

### 3. Reference

---

**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` is 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(BadCustomName.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()**  
**pwmoutput→enabledAtPowerOn()**  
**pwmoutput.get\_enabledAtPowerOn( )**

---

**YPwmOutput**

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 occurred 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 occurred while using the PWM object

`pwmoutput→get_frequency()`  
`pwmoutput→frequency()`  
`pwmoutput.get_frequency()`

---

YPwmOutput

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()**  
**pwmoutput→functionId()**  
**pwmoutput.get\_functionId()**

---

**YPwmOutput**

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()**  
**pwmoutput→logicalName()**  
**pwmoutput.get\_logicalName()**

---

**YPwmOutput**

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()**  
**pwmoutput→pulseDuration()**  
**pwmoutput.get\_pulseDuration()**

---

**YPwmOutput**

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()**  
**pwmoutput→userData()**  
**pwmoutput.getUserData( )**

---

**YPwmOutput**

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**pwmoutput→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()**  
**pwmoutput→setEnabled()**  
**pwmoutput.set\_enabled()**

---

**YPwmOutput**

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()**  
**pwmoutput→setFrequency()**  
**pwmoutput.set\_frequency( )**

---

**YPwmOutput**

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()**  
**pwmoutput→setPulseDuration()**  
**pwmoutput.set\_pulseDuration( )**

---

**YPwmOutput**

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()**  
**pwmoutput→setUserData()**  
**pwmoutput.set\_userdata( )**

---

**YPwmOutput**

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');
c++	#include "yocto_pwmpowersource.h"
m	#import "yocto_pwmpowersource.h"
pas	uses yocto_pwmpowersource;
vb	yocto_pwmpowersource.vb
cs	yocto_pwmpowersource.cs
java	import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py	from yocto_pwmpowersource import *

### Global functions

#### yFindPwmPowerSource(func)

Retrieves a voltage source for a given identifier.

#### yFirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

### YPwmPowerSource methods

#### pwmpowersource→describe()

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

#### pwmpowersource→get\_advertisedValue()

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

#### pwmpowersource→get\_errorMessage()

Returns the error message of the latest error with the voltage source.

#### pwmpowersource→get\_errorType()

Returns the numerical error code of the latest error with the voltage source.

#### pwmpowersource→get\_friendlyName()

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

#### pwmpowersource→get\_functionDescriptor()

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

#### pwmpowersource→get\_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

#### pwmpowersource→get\_hardwareId()

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

#### pwmpowersource→get\_logicalName()

Returns the logical name of the voltage source.

#### pwmpowersource→get\_module()

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

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

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

**pwmpowersource→get\_powerMode()**

Returns the selected power source for the PWM on the same device

**pwmpowersource→get\_userData()**

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

**pwmpowersource→isOnline()**

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

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

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

**pwmpowersource→load(msValidity)**

Preloads the voltage source cache with a specified validity duration.

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

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

**pwmpowersource→nextPwmPowerSource()**

Continues the enumeration of Voltage sources started using yFirstPwmPowerSource( ).

**pwmpowersource→registerValueCallback(callback)**

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

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

Changes the logical name of the voltage source.

**pwmpowersource→set\_powerMode(newval)**

Changes the PWM power source.

**pwmpowersource→set\_userData(data)**

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

**pwmpowersource→wait\_async(callback, context)**

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

**YPwmPowerSource.FindPwmPowerSource()  
yFindPwmPowerSource()  
YPwmPowerSource.FindPwmPowerSource( )**

**YPwmPowerSource**

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()  
yFirstPwmPowerSource()  
YPwmPowerSource.FirstPwmPowerSource( )**

---

**YPwmPowerSource**

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` is 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(BadCustomName.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 occurred 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 occurred while using the voltage source object

**pwmpowersource→get\_friendlyName()**

**YPwmPowerSource**

**pwmpowersource→friendlyName()**

**pwmpowersource.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

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

---

**pwmpowersource→get\_functionDescriptor()****YPwmPowerSource****pwmpowersource→functionDescriptor()****pwmpowersource.get\_functionDescriptor()**

---

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

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()**  
**pwmpowersource→module()**  
**pwmpowersource.get\_module()**

---

**YPwmPowerSource**

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**pwmpowersource→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()**  
**pwmpowersource→setPowerMode()**  
**pwmpowersource.set\_powerMode( )**

**YPwmPowerSource**

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	<code>&lt;script type='text/javascript' src='yocto_gyro.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YGyro = yoctolib.YGyro;</code>
php	<code>require_once('yocto_gyro.php');</code>
c++	<code>#include "yocto_gyro.h"</code>
m	<code>#import "yocto_gyro.h"</code>
pas	<code>uses yocto_gyro;</code>
vb	<code>yocto_gyro.vb</code>
cs	<code>yocto_gyro.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YGyro;</code>
py	<code>from yocto_gyro import *</code>

### 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 <code>SERIAL.FUNCTIONID</code> .
<b>qt→get_highestValue()</b>	Returns the maximal value observed for the value since the device was started.
<b>qt→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>qt→get_logicalName()</b>	Returns the logical name of the quaternion component.
<b>qt→get_lowestValue()</b>	Returns the minimal value observed for the value since the device was started.
<b>qt→get_module()</b>	Gets the <code>YModule</code> object for the device on which the function is located.
<b>qt→get_module_async(callback, context)</b>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<b>qt→get_recordedData(startTime, endTime)</b>	Retrieves a <code>DataSet</code> object holding historical data for this sensor, for a specified time interval.
<b>qt→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>qt→get_resolution()</b>	Returns the resolution of the measured values.
<b>qt→get_unit()</b>	Returns the measuring unit for the value.
<b>qt→get_userData()</b>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b>qt→isOnline()</b>	Checks if the quaternion component is currently reachable, without raising any error.
<b>qt→isOnline_async(callback, context)</b>	Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).
<b>qt→load(msValidity)</b>	Preloads the quaternion component cache with a specified validity duration.
<b>qt→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code> .
<b>qt→load_async(msValidity, callback, context)</b>	Preloads the quaternion component cache with a specified validity duration (asynchronous version).
<b>qt→nextQt()</b>	Continues the enumeration of quaternion components started using <code>yFirstQt()</code> .
<b>qt→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>qt→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>qt→set_highestValue(newval)</b>	Changes the recorded maximal value observed.
<b>qt→set_logFrequency(newval)</b>	Changes the datalogger recording frequency for this function.
<b>qt→set_logicalName(newval)</b>	

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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(BadCustomName.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 occurred 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 occurred while using the quaternion component object



---

**qt→get\_friendlyName()****YQt****qt→friendlyName()qt.get\_friendlyName()**

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

**qt→get\_functionDescriptor()****YQt****qt→functionDescriptor()****qt.get\_functionDescriptor()**

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

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 on-line.

**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 traffic for instance.

**Parameters :**

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

**Returns :**

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



**qt→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');
c++	#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
<b>yFindRealTimeClock(func)</b> Retrieves a clock for a given identifier.
<b>yFirstRealTimeClock()</b> Starts the enumeration of clocks currently accessible.
YRealTimeClock methods
<b>realtimeclock→describe()</b> Returns a short text that describes unambiguously the instance of the clock in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.
<b>realtimeclock→get_advertisedValue()</b> Returns the current value of the clock (no more than 6 characters).
<b>realtimeclock→get_dateTime()</b> Returns the current time in the form "YYYY/MM/DD hh:mm:ss"
<b>realtimeclock→get_errorMessage()</b> Returns the error message of the latest error with the clock.
<b>realtimeclock→get_errorType()</b> Returns the numerical error code of the latest error with the clock.
<b>realtimeclock→get_friendlyName()</b> Returns a global identifier of the clock in the format MODULE_NAME . FUNCTION_NAME.
<b>realtimeclock→get_functionDescriptor()</b> Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>realtimeclock→get_functionId()</b> Returns the hardware identifier of the clock, without reference to the module.
<b>realtimeclock→get_hardwareId()</b> Returns the unique hardware identifier of the clock in the form SERIAL . FUNCTIONID.
<b>realtimeclock→get_logicalName()</b> Returns the logical name of the clock.
<b>realtimeclock→get_module()</b>

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` is 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(BadCustomName.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 occurred 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 occurred while using the clock object

---

**realtimeclock→get\_friendlyName()****YRealTimeClock****realtimeclock→friendlyName()****realtimeclock.get\_friendlyName()**

---

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

**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 example: `MyCustomName.relay1`)

**Returns :**

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

**realtimeclock→get\_functionDescriptor()**

**YRealTimeClock**

**realtimeclock→functionDescriptor()**

**realtimeclock.get\_functionDescriptor()**

---

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

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()**  
**realtimeclock→userData()**  
**realtimeclock.get\_userdata()**

---

**YRealTimeClock**

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\_UTC\_OFFSET\_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 traffic for instance.

**Parameters :**

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

**Returns :**

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



---

**realtimeclock→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_reframe.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YRefFrame = yoctolib.YRefFrame;
php	require_once('yocto_reframe.php');
c++	#include "yocto_reframe.h"
m	#import "yocto_reframe.h"
pas	uses yocto_reframe;
vb	yocto_reframe.vb
cs	yocto_reframe.cs
java	import com.yoctopuce.YoctoAPI.YRefFrame;
py	from yocto_reframe import *

### Global functions

#### yFindRefFrame(func)

Retrieves a reference frame for a given identifier.

#### yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

### YRefFrame methods

#### reframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

#### reframe→describe()

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

#### reframe→get\_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

#### reframe→get\_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

#### reframe→get\_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

#### reframe→get\_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

#### reframe→get\_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

#### reframe→get\_advertisedValue()

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

#### reframe→get\_bearing()

Returns the reference bearing used by the compass.

**reiframe→get\_errorMessage()**

Returns the error message of the latest error with the reference frame.

**reiframe→get\_errorType()**

Returns the numerical error code of the latest error with the reference frame.

**reiframe→get\_friendlyName()**

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

**reiframe→get\_functionDescriptor()**

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

**reiframe→get\_functionId()**

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

**reiframe→get\_hardwareId()**

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

**reiframe→get\_logicalName()**

Returns the logical name of the reference frame.

**reiframe→get\_module()**

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

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

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

**reiframe→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.

**reiframe→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.

**reiframe→get\_userData()**

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

**reiframe→isOnline()**

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

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

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

**reiframe→load(msValidity)**

Preloads the reference frame cache with a specified validity duration.

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

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

**reiframe→more3DCalibration()**

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

**reiframe→nextRefFrame()**

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

**reiframe→registerValueCallback(callback)**

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

**reiframe→save3DCalibration()**

Applies the sensors tridimensional calibration parameters that have just been computed.

**reiframe→set\_bearing(newval)**

Changes the reference bearing used by the compass.

**reiframe→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` is 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(BadCustomName.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 occurred 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 occurred while using the reference frame object

---

**refframe→get\_friendlyName()****YRefFrame****refframe→friendlyName()****refframe.get\_friendlyName()**

---

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

**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 example: `MyCustomName.relay1`)

**Returns :**

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

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

**refframe→get\_functionDescriptor()**

**YRefFrame**

**refframe→functionDescriptor()**

**refframe.get\_functionDescriptor()**

---

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

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**refframe→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()**  
**refframe**→**setLogicalName()**  
**refframe.set\_logicalName()**

---

**YRefFrame**

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	<code>&lt;script type='text/javascript' src='yocto_relay.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YRelay = yoctolib.YRelay;</code>
php	<code>require_once('yocto_relay.php');</code>
c++	<code>#include "yocto_relay.h"</code>
m	<code>#import "yocto_relay.h"</code>
pas	<code>uses yocto_relay;</code>
vb	<code>yocto_relay.vb</code>
cs	<code>yocto_relay.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YRelay;</code>
py	<code>from yocto_relay import *</code>

### 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 milliseconds

**ms\_duration** pulse duration, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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



---

**relay→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` is 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(BadCustomName.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.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 occurred 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 occurred while using the relay object

**relay**→**get\_friendlyName()**

**YRelay**

**relay**→**friendlyName()****relay.get\_friendlyName()**

---

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

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 on-line.

**Returns :**

an instance of YModule

**relay**→**get\_output()**

**YRelay**

**relay**→**output()**`relay.get_output( )`

---

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

`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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**relay**→**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 milliseconds

**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()**  
**relay**→**setLogicalName()**  
**relay.set\_logicalName()**

**YRelay**

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()

### 3. Reference

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**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(BadCustomName.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 occurred 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 occurred while using the sensor object

---

**sensor**→**get\_friendlyName()**  
**sensor**→**friendlyName()**  
**sensor.get\_friendlyName()**

---

**YSensor**

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

**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()**  
**sensor**→**functionDescriptor()**  
**sensor.get\_functionDescriptor()**

---

**YSensor**

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**sensor→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	<code>&lt;script type='text/javascript' src='yocto_servo.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YServo = yoctolib.YServo;</code>
php	<code>require_once('yocto_servo.php');</code>
c++	<code>#include "yocto_servo.h"</code>
m	<code>#import "yocto_servo.h"</code>
pas	<code>uses yocto_servo;</code>
vb	<code>yocto_servo.vb</code>
cs	<code>yocto_servo.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YServo;</code>
py	<code>from yocto_servo import *</code>

### 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)**

### 3. Reference

---

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` is 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(BadCustomName.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.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 occurred 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 occurred while using the servo object



---

**servo→get\_friendlyName()****YServo****servo→friendlyName()**`servo.get_friendlyName()`

---

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

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 example: `MyCustomName.relay1`)

**Returns :**

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

**servo→get\_functionDescriptor()**

**YServo**

**servo→functionDescriptor()**

**servo.get\_functionDescriptor()**

---

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**servo→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');
c++	#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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**temperature→describe()**`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` is 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(BadCustomName.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 occurred 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 occurred while using the temperature sensor object

**temperature**→**get\_friendlyName()****YTemperature****temperature**→**friendlyName()****temperature.get\_friendlyName()**

---

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

**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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**temperature→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()**  
**temperature**→**setHighestValue()**  
**temperature.set\_highestValue()**

---

**YTemperature**

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()**  
**temperature**→**setResolution()**  
**temperature.set\_resolution()**

**YTemperature**

---

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');
c++	#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.

### 3. Reference

---

**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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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` is 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(BadCustomName.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 occurred 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 occurred while using the tilt sensor object

**tilt**→**get\_friendlyName()****YTilt****tilt**→**friendlyName()****tilt.get\_friendlyName()**

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

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**tilt→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');
c++	#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()

### 3. Reference

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**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` is 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(BadCustomName.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 occurred 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 occurred while using the Volatile Organic Compound sensor object

**voc**→**get\_friendlyName()****YVoc****voc**→**friendlyName()****voc.get\_friendlyName()**

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

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()**  
**voc→reportFrequency()**  
**voc.get\_reportFrequency( )**

**YVoc**

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**voc→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()**  
**voc**→**setReportFrequency()**  
**voc.set\_reportFrequency( )**

**YVoc**

---

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.

**YVoc**  
**voc**→**set\_resolution()****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	<code>&lt;script type='text/javascript' src='yocto_voltage.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YVoltage = yoctolib.YVoltage;</code>
php	<code>require_once('yocto_voltage.php');</code>
c++	<code>#include "yocto_voltage.h"</code>
m	<code>#import "yocto_voltage.h"</code>
pas	<code>uses yocto_voltage;</code>
vb	<code>yocto_voltage.vb</code>
cs	<code>yocto_voltage.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YVoltage;</code>
py	<code>from yocto_voltage import *</code>

### 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.

### 3. Reference

---

**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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**voltage→describe()**`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` is 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(BadCustomName.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()**  
**voltage**→**currentValue()**  
**voltage.get\_currentValue()**

---

**YVoltage**

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 occurred 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 occurred while using the voltage sensor object

**voltage→get\_friendlyName()**

**YVoltage**

**voltage→friendlyName()**

**voltage.get\_friendlyName()**

---

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

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**voltage→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()**  
**voltage**→**setResolution()**  
**voltage.set\_resolution()**

**YVoltage**

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	<code>&lt;script type='text/javascript' src='yocto_vsource.js'&gt;&lt;/script&gt;</code>
php	<code>require_once('yocto_vsource.php');</code>
c++	<code>#include "yocto_vsource.h"</code>
m	<code>#import "yocto_vsource.h"</code>
pas	<code>uses yocto_vsource;</code>
vb	<code>yocto_vsource.vb</code>
cs	<code>yocto_vsource.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YVSource;</code>
py	<code>from yocto_vsource import *</code>

Global functions
<b>yFindVSource(func)</b> Retrieves a voltage source for a given identifier.
<b>yFirstVSource()</b> Starts the enumeration of voltage sources currently accessible.
YVSource methods
<b>vsource→describe()</b> Returns a short text that describes the function in the form TYPE ( NAME ) =SERIAL . FUNCTIONID.
<b>vsource→get_advertisedValue()</b> Returns the current value of the voltage source (no more than 6 characters).
<b>vsource→get_errorMessage()</b> Returns the error message of the latest error with this function.
<b>vsource→get_errorType()</b> Returns the numerical error code of the latest error with this function.
<b>vsource→get_extPowerFailure()</b> Returns true if external power supply voltage is too low.
<b>vsource→get_failure()</b> Returns true if the module is in failure mode.
<b>vsource→get_friendlyName()</b> Returns a global identifier of the function in the format MODULE_NAME . FUNCTION_NAME.
<b>vsource→get_functionDescriptor()</b> Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>vsource→get_functionId()</b> Returns the hardware identifier of the function, without reference to the module.
<b>vsource→get_hardwareId()</b> Returns the unique hardware identifier of the function in the form SERIAL . FUNCTIONID.
<b>vsource→get_logicalName()</b> Returns the logical name of the voltage source.
<b>vsource→get_module()</b> Gets the YModule object for the device on which the function is located.
<b>vsource→get_module_async(callback, context)</b>

### 3. Reference

	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<b><code>vsource→get_overCurrent()</code></b>	Returns true if the appliance connected to the device is too greedy .
<b><code>vsource→get_overHeat()</code></b>	Returns TRUE if the module is overheating.
<b><code>vsource→get_overLoad()</code></b>	Returns true if the device is not able to maintaint the requested voltage output .
<b><code>vsource→get_regulationFailure()</code></b>	Returns true if the voltage output is too high regarding the requested voltage .
<b><code>vsource→get_unit()</code></b>	Returns the measuring unit for the voltage.
<b><code>vsource→get_userData()</code></b>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b><code>vsource→get_voltage()</code></b>	Returns the voltage output command (mV)
<b><code>vsource→isOnline()</code></b>	Checks if the function is currently reachable, without raising any error.
<b><code>vsource→isOnline_async(callback, context)</code></b>	Checks if the function is currently reachable, without raising any error (asynchronous version).
<b><code>vsource→load(msValidity)</code></b>	Preloads the function cache with a specified validity duration.
<b><code>vsource→load_async(msValidity, callback, context)</code></b>	Preloads the function cache with a specified validity duration (asynchronous version).
<b><code>vsource→nextVSource()</code></b>	Continues the enumeration of voltage sources started using <code>yFirstVSource()</code> .
<b><code>vsource→pulse(voltage, ms_duration)</code></b>	Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.
<b><code>vsource→registerValueCallback(callback)</code></b>	Registers the callback function that is invoked on every change of advertised value.
<b><code>vsource→set_logicalName(newval)</code></b>	Changes the logical name of the voltage source.
<b><code>vsource→set_userData(data)</code></b>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<b><code>vsource→set_voltage(newval)</code></b>	Tunes the device output voltage (milliVolts).
<b><code>vsource→voltageMove(target, ms_duration)</code></b>	Performs a smooth move at constant speed toward a given value.
<b><code>vsource→wait_async(callback, context)</code></b>	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 <b>yFindVSource</b> ( <b>func</b> )
php	function <b>yFindVSource</b> ( <b>\$func</b> )
cpp	YVSource* <b>yFindVSource</b> ( const string& <b>func</b> )
m	YVSource* <b>yFindVSource</b> ( NSString* <b>func</b> )
pas	function <b>yFindVSource</b> ( <b>func</b> : string): TYVSource
vb	function <b>yFindVSource</b> ( ByVal <b>func</b> As String) As YVSource
cs	YVSource <b>FindVSource</b> ( string <b>func</b> )
java	YVSource <b>FindVSource</b> ( String <b>func</b> )
py	def <b>FindVSource</b> ( <b>func</b> )

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 <b>yFirstVSource()</b>
php	function <b>yFirstVSource()</b>
cpp	YVSource* <b>yFirstVSource()</b>
m	YVSource* <b>yFirstVSource()</b>
pas	function <b>yFirstVSource()</b> : TYVSource
vb	function <b>yFirstVSource()</b> As YVSource
cs	YVSource <b>FirstVSource()</b>
java	YVSource <b>FirstVSource()</b>
py	def <b>FirstVSource()</b>

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(BadCustomName.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 <b>get_advertisedValue()</b>
php	function <b>get_advertisedValue()</b>
cpp	string <b>get_advertisedValue()</b>
m	-(NSString*) advertisedValue
pas	function <b>get_advertisedValue()</b> : string
vb	function <b>get_advertisedValue()</b> As String
cs	string <b>get_advertisedValue()</b>
java	String <b>get_advertisedValue()</b>
py	def <b>get_advertisedValue()</b>
cmd	YVSource <b>target</b> <b>get_advertisedValue</b>

**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.errorMessage()**


---

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

String **get\_errorMessage()**

**vsource**→**get\_errorMessage()****vsource**→**errorMessage()****vsource.errorMessage()**


---

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

js	function <b>get_errorMessage()</b>
php	function <b>get_errorMessage()</b>
cpp	string <b>get_errorMessage()</b>
m	-(NSString*) errorMessage
pas	function <b>get_errorMessage()</b> : string
vb	function <b>get_errorMessage()</b> As String
cs	string <b>get_errorMessage()</b>
java	String <b>get_errorMessage()</b>
py	def <b>get_errorMessage()</b>

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

**Returns :**

a string corresponding to the latest error message that occurred while using this function object

**vsource**→**get\_errorType()****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 <b>get_errorType</b> ( )
php	function <b>get_errorType</b> ( )
cpp	YRETCODE <b>get_errorType</b> ( )
pas	function <b>get_errorType</b> ( ): YRETCODE
vb	function <b>get_errorType</b> ( ) As YRETCODE
cs	YRETCODE <b>get_errorType</b> ( )
java	int <b>get_errorType</b> ( )
py	def <b>get_errorType</b> ( )

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

**Returns :**

a number corresponding to the code of the latest error that occurred while using this function object

**vsources**→**get\_extPowerFailure()****YVSource****vsources**→**extPowerFailure()****vsources.get\_extPowerFailure()**

Returns true if external power supply voltage is too low.

**int** **get\_extPowerFailure()****vsources**→**get\_extPowerFailure()****vsources**→**extPowerFailure()****vsources.get\_extPowerFailure()**

Returns true if external power supply voltage is too low.

js	function <b>get_extPowerFailure()</b>
php	function <b>get_extPowerFailure()</b>
cpp	Y_EXTPOWERFAILURE_enum <b>get_extPowerFailure()</b>
m	-(Y_EXTPOWERFAILURE_enum) extPowerFailure
pas	function <b>get_extPowerFailure()</b> : Integer
vb	function <b>get_extPowerFailure()</b> As Integer
cs	int <b>get_extPowerFailure()</b>
java	int <b>get_extPowerFailure()</b>
py	def <b>get_extPowerFailure()</b>
cmd	YVSource <b>target</b> <b>get_extPowerFailure</b>

**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 <b>get_failure()</b>
php	function <b>get_failure()</b>
cpp	Y_FAILURE_enum <b>get_failure()</b>
m	-(Y_FAILURE_enum) failure
pas	function <b>get_failure()</b> : Integer
vb	function <b>get_failure()</b> As Integer
cs	int <b>get_failure()</b>
java	int <b>get_failure()</b>
py	def <b>get_failure()</b>
cmd	YVSource <b>target</b> <b>get_failure</b>

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 example: `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`.

**vsSource→get\_functionDescriptor()****YVSource****vsSource→functionDescriptor()****vsSource.get\_vsSourceDescriptor()**


---

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

String **get\_functionDescriptor()**

**vsSource→get\_functionDescriptor()****vsSource→functionDescriptor()vsSource.get\_vsSourceDescriptor()**


---

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

js	function <b>get_functionDescriptor()</b>
php	function <b>get_functionDescriptor()</b>
cpp	YFUN_DESCR <b>get_functionDescriptor()</b>
m	-(YFUN_DESCR) functionDescriptor
pas	function <b>get_functionDescriptor()</b> : YFUN_DESCR
vb	function <b>get_functionDescriptor()</b> As YFUN_DESCR
cs	YFUN_DESCR <b>get_functionDescriptor()</b>
java	String <b>get_functionDescriptor()</b>
py	def <b>get_functionDescriptor()</b>

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`.

**vsources**→**get\_hardwareId()**

**YVSource**

**vsources**→**hardwareId()****vsources.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()**  
**vsource**→**logicalName()**  
**vsource.get\_logicalName()**

YVSource

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 <b>get_logicalName()</b>
php	function <b>get_logicalName()</b>
cpp	string <b>get_logicalName()</b>
m	-(NSString*) logicalName
pas	function <b>get_logicalName()</b> : string
vb	function <b>get_logicalName()</b> As String
cs	string <b>get_logicalName()</b>
java	String <b>get_logicalName()</b>
py	def <b>get_logicalName()</b>
cmd	YVSource <b>target</b> <b>get_logicalName</b>

#### Returns :

a string corresponding to the logical name of the voltage source

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

**vsourc**→**get\_module()****YVSource****vsourc**→**module()****vsourc.get\_module()**

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

YModule **get\_module()****vsourc**→**get\_module()****vsourc**→**module()****vsourc.get\_module()**

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

<code>js</code>	<code>function <b>get_module()</b></code>
<code>php</code>	<code>function <b>get_module()</b></code>
<code>cpp</code>	<code>YModule * <b>get_module()</b></code>
<code>m</code>	<code>-(YModule*) module</code>
<code>pas</code>	<code>function <b>get_module()</b> : TModule</code>
<code>vb</code>	<code>function <b>get_module()</b> As YModule</code>
<code>cs</code>	<code>YModule <b>get_module()</b></code>
<code>java</code>	<code>YModule <b>get_module()</b></code>
<code>py</code>	<code>def <b>get_module()</b></code>

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 .

<b>js</b>	<b>function</b> <b>get_overCurrent( )</b>
<b>php</b>	<b>function</b> <b>get_overCurrent( )</b>
<b>cpp</b>	<b>Y_OVERCURRENT_enum</b> <b>get_overCurrent( )</b>
<b>m</b>	<b>-(Y_OVERCURRENT_enum)</b> <b>overCurrent</b>
<b>pas</b>	<b>function</b> <b>get_overCurrent( )</b> : Integer
<b>vb</b>	<b>function</b> <b>get_overCurrent( )</b> As Integer
<b>cs</b>	<b>int</b> <b>get_overCurrent( )</b>
<b>java</b>	<b>int</b> <b>get_overCurrent( )</b>
<b>py</b>	<b>def</b> <b>get_overCurrent( )</b>
<b>cmd</b>	<b>YVSource</b> <b>target</b> <b>get_overCurrent</b>

**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**.

**vsources**→**get\_overHeat()****YVSource****vsources**→**overHeat()****vsources.get\_overHeat( )**

Returns TRUE if the module is overheating.

**int** **get\_overHeat( )****vsources**→**get\_overHeat()****vsources**→**overHeat()****vsources.get\_overHeat( )**

Returns TRUE if the module is overheating.

js	function <b>get_overHeat( )</b>
php	function <b>get_overHeat( )</b>
cpp	Y_OVERHEAT_enum <b>get_overHeat( )</b>
m	-(Y_OVERHEAT_enum) overHeat
pas	function <b>get_overHeat( )</b> : Integer
vb	function <b>get_overHeat( )</b> As Integer
cs	int <b>get_overHeat( )</b>
java	int <b>get_overHeat( )</b>
py	def <b>get_overHeat( )</b>
cmd	YVSource <b>target</b> <b>get_overHeat</b>

**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 .

<b>js</b>	<b>function</b> <b>get_overLoad( )</b>
<b>php</b>	<b>function</b> <b>get_overLoad( )</b>
<b>cpp</b>	<b>Y_OVERLOAD_enum</b> <b>get_overLoad( )</b>
<b>m</b>	<b>-(Y_OVERLOAD_enum)</b> <b>overLoad</b>
<b>pas</b>	<b>function</b> <b>get_overLoad( )</b> : <b>Integer</b>
<b>vb</b>	<b>function</b> <b>get_overLoad( )</b> <b>As Integer</b>
<b>cs</b>	<b>int</b> <b>get_overLoad( )</b>
<b>java</b>	<b>int</b> <b>get_overLoad( )</b>
<b>py</b>	<b>def</b> <b>get_overLoad( )</b>
<b>cmd</b>	<b>YVSource</b> <b>target</b> <b>get_overLoad</b>

**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.

**vsSource**→**get\_regulationFailure()****YVSource****vsSource**→**regulationFailure()****vsSource.get\_regulationFailure()**


---

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

```
int get_regulationFailure( )
```

**vsSource**→**get\_regulationFailure()****vsSource**→**regulationFailure()****vsSource.get\_regulationFailure()**


---

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

js	function <b>get_regulationFailure</b> ( )
php	function <b>get_regulationFailure</b> ( )
cpp	Y_REGULATIONFAILURE_enum <b>get_regulationFailure</b> ( )
m	-(Y_REGULATIONFAILURE_enum) regulationFailure
pas	function <b>get_regulationFailure</b> ( ): Integer
vb	function <b>get_regulationFailure</b> ( ) As Integer
cs	int <b>get_regulationFailure</b> ( )
java	int <b>get_regulationFailure</b> ( )
py	def <b>get_regulationFailure</b> ( )
cmd	YVSource <b>target</b> <b>get_regulationFailure</b>

**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 <b>get_unit()</b>
php	function <b>get_unit()</b>
cpp	string <b>get_unit()</b>
m	-(NSString*) unit
pas	function <b>get_unit()</b> : string
vb	function <b>get_unit()</b> As String
cs	string <b>get_unit()</b>
java	String <b>get_unit()</b>
py	def <b>get_unit()</b>
cmd	YVSource <b>target</b> <b>get_unit</b>

**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 <b>get_userData( )</b>
php	function <b>get_userData( )</b>
cpp	void * <b>get_userData( )</b>
m	-(void*) userData
pas	function <b>get_userData( )</b> : Tobject
vb	function <b>get_userData( )</b> As Object
cs	object <b>get_userData( )</b>
java	Object <b>get_userData( )</b>
py	def <b>get_userData( )</b>

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 <b>get_voltage</b> ( )
php	function <b>get_voltage</b> ( )
cpp	int <b>get_voltage</b> ( )
m	-(int) voltage
pas	function <b>get_voltage</b> ( ): LongInt
vb	function <b>get_voltage</b> ( ) As Integer
cs	int <b>get_voltage</b> ( )
java	int <b>get_voltage</b> ( )
py	def <b>get_voltage</b> ( )

**Returns :**

an integer corresponding to the voltage output command (mV)

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

**vsourc**→**isOnline()****vsourc.isOnline( )****YVSource**

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

boolean **isOnline( )**

**vsourc**→**isOnline()****vsourc.isOnline( )**

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

js	function <b>isOnline( )</b>
php	function <b>isOnline( )</b>
cpp	bool <b>isOnline( )</b>
m	-(BOOL) <b>isOnline</b>
pas	function <b>isOnline( )</b> : boolean
vb	function <b>isOnline( )</b> As Boolean
cs	bool <b>isOnline( )</b>
java	boolean <b>isOnline( )</b>
py	def <b>isOnline( )</b>

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**vs**source→nextVSource()**vs**source.nextVSource( )

YVSource

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

YVSource nextVSource( )

**vs**source→nextVSource()**vs**source.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 milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**vsources.registerValueCallback()****YVSource****vsources.registerValueCallback()**

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

```
void registerValueCallback( UpdateCallback callback)
```

**vsources.registerValueCallback()vsources.registerValueCallback()**

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

js	function registerValueCallback( callback)
php	function registerValueCallback( \$callback)
cpp	void registerValueCallback( YDisplayUpdateCallback callback)
pas	procedure registerValueCallback( callback: TGenericUpdateCallback)
vb	procedure registerValueCallback( ByVal callback As GenericUpdateCallback)
cs	void registerValueCallback( UpdateCallback callback)
java	void registerValueCallback( UpdateCallback callback)
py	def registerValueCallback( callback)
m	-(void) registerValueCallback : (YFunctionUpdateCallback) callback

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

**Parameters :**

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

**vsource**→**set\_logicalName()**  
**vsource**→**setLogicalName()**  
**vsource.set\_logicalName()**

YVSource

Changes the logical name of the voltage source.

```
int set_logicalName( String newval)
```

**vsource**→**set\_logicalName()**  
**vsource**→**setLogicalName()****vsource.set\_logicalName()**

Changes the logical name of the voltage source.

js	function <b>set_logicalName</b> ( newval)
php	function <b>set_logicalName</b> ( \$newval)
cpp	int <b>set_logicalName</b> ( const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function <b>set_logicalName</b> ( newval: string): integer
vb	function <b>set_logicalName</b> ( ByVal newval As String) As Integer
cs	int <b>set_logicalName</b> ( string newval)
java	int <b>set_logicalName</b> ( String newval)
py	def <b>set_logicalName</b> ( newval)
cmd	YVSource <b>target set_logicalName newval</b>

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 <b>set_userdata</b> ( <b>data</b> )
php	function <b>set_userdata</b> ( <b>\$data</b> )
cpp	void <b>set_userdata</b> ( void* <b>data</b> )
m	-(void) setUserData : (void*) <b>data</b>
pas	procedure <b>set_userdata</b> ( <b>data</b> : Tobject)
vb	procedure <b>set_userdata</b> ( ByVal <b>data</b> As Object)
cs	void <b>set_userdata</b> ( object <b>data</b> )
java	void <b>set_userdata</b> ( Object <b>data</b> )
py	def <b>set_userdata</b> ( <b>data</b> )

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 <b>set_voltage</b> ( <b>newval</b> )
php	function <b>set_voltage</b> ( <b>\$newval</b> )
cpp	int <b>set_voltage</b> ( int <b>newval</b> )
m	-(int) setVoltage : (int) <b>newval</b>
pas	function <b>set_voltage</b> ( <b>newval</b> : LongInt): integer
vb	function <b>set_voltage</b> ( ByVal <b>newval</b> As Integer) As Integer
cs	int <b>set_voltage</b> ( int <b>newval</b> )
java	int <b>set_voltage</b> ( int <b>newval</b> )
py	def <b>set_voltage</b> ( <b>newval</b> )
cmd	YVSource <b>target set_voltage newval</b>

**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');
c++	#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()

### 3. Reference

	Returns the next scheduled wake up date/time (UNIX format)
<b>wakeupmonitor→get_powerDuration()</b>	Returns the maximal wake up time (in seconds) before automatically going to sleep.
<b>wakeupmonitor→get_sleepCountdown()</b>	Returns the delay before the next sleep period.
<b>wakeupmonitor→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>wakeupmonitor→get_wakeUpReason()</b>	Returns the latest wake up reason.
<b>wakeupmonitor→get_wakeUpState()</b>	Returns the current state of the monitor
<b>wakeupmonitor→isOnline()</b>	Checks if the monitor is currently reachable, without raising any error.
<b>wakeupmonitor→isOnline_async(callback, context)</b>	Checks if the monitor is currently reachable, without raising any error (asynchronous version).
<b>wakeupmonitor→load(msValidity)</b>	Preloads the monitor cache with a specified validity duration.
<b>wakeupmonitor→load_async(msValidity, callback, context)</b>	Preloads the monitor cache with a specified validity duration (asynchronous version).
<b>wakeupmonitor→nextWakeUpMonitor()</b>	Continues the enumeration of monitors started using <code>yFirstWakeUpMonitor()</code> .
<b>wakeupmonitor→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>wakeupmonitor→resetSleepCountDown()</b>	Resets the sleep countdown.
<b>wakeupmonitor→set_logicalName(newval)</b>	Changes the logical name of the monitor.
<b>wakeupmonitor→set_nextWakeUp(newval)</b>	Changes the days of the week when a wake up must take place.
<b>wakeupmonitor→set_powerDuration(newval)</b>	Changes the maximal wake up time (seconds) before automatically going to sleep.
<b>wakeupmonitor→set_sleepCountdown(newval)</b>	Changes the delay before the next sleep period.
<b>wakeupmonitor→set_userData(data)</b>	Stores a user context provided as argument in the userData attribute of the function.
<b>wakeupmonitor→sleep(secBeforeSleep)</b>	Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.
<b>wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep)</b>	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.
<b>wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep)</b>	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.
<b>wakeupmonitor→wait_async(callback, context)</b>	



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` is 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(BadCustomName.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 occurred 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 occurred while using the monitor object

**wakeupmonitor→get\_friendlyName()**

**YWakeUpMonitor**

**wakeupmonitor→friendlyName()**

**wakeupmonitor.get\_friendlyName()**

---

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

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\_SLEEP\_COUNTDOWN\_INVALID.



---

**wakeupmonitor→get\_userdata()****YWakeUpMonitor****wakeupmonitor→userData()****wakeupmonitor.getUserData()**

---

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**wakeupmonitor→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()**  
**wakeupmonitor**→**setLogicalName()**  
**wakeupmonitor.set\_logicalName()**

**YWakeUpMonitor**

---

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()**  
**wakeupmonitor**→**setSleepCountdown()**  
**wakeupmonitor.set\_sleepCountdown( )**

**YWakeUpMonitor**

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()**  
**wakeupmonitor.wakeUp( )**

---

**YWakeUpMonitor**

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');
c++	#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
<b>yFindWakeUpSchedule(func)</b> Retrieves a wake up schedule for a given identifier.
<b>yFirstWakeUpSchedule()</b> Starts the enumeration of wake up schedules currently accessible.
YWakeUpSchedule methods
<b>wakeupschedule→describe()</b> Returns a short text that describes unambiguously the instance of the wake up schedule in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.
<b>wakeupschedule→get_advertisedValue()</b> Returns the current value of the wake up schedule (no more than 6 characters).
<b>wakeupschedule→get_errorMessage()</b> Returns the error message of the latest error with the wake up schedule.
<b>wakeupschedule→get_errorType()</b> Returns the numerical error code of the latest error with the wake up schedule.
<b>wakeupschedule→get_friendlyName()</b> Returns a global identifier of the wake up schedule in the format MODULE_NAME . FUNCTION_NAME.
<b>wakeupschedule→get_functionDescriptor()</b> Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>wakeupschedule→get_functionId()</b> Returns the hardware identifier of the wake up schedule, without reference to the module.
<b>wakeupschedule→get_hardwareId()</b> Returns the unique hardware identifier of the wake up schedule in the form SERIAL . FUNCTIONID.
<b>wakeupschedule→get_hours()</b> Returns the hours scheduled for wake up.
<b>wakeupschedule→get_logicalName()</b> Returns the logical name of the wake up schedule.
<b>wakeupschedule→get_minutes()</b> Returns all the minutes of each hour that are scheduled for wake up.
<b>wakeupschedule→get_minutesA()</b>

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 interval of 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 occurrence

#### **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.

### 3. Reference

---

**wakeupschedule→set\_weekDays(newval)**

Changes the days of the week when a wake up must take place.

**wakeupschedule→wait\_async(callback, context)**

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

## YWakeUpSchedule.FindWakeUpSchedule() yFindWakeUpSchedule()

## YWakeUpSchedule

### 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()  
yFirstWakeUpSchedule()  
YWakeUpSchedule.FirstWakeUpSchedule( )**

---

**YWakeUpSchedule**

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` is 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(BadCustomName.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 occurred 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 occurred while using the wake up schedule object

---

**wakeupschedule→get\_friendlyName()****YWakeUpSchedule****wakeupschedule→friendlyName()****wakeupschedule.get\_friendlyName()**

---

Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

**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()**  
**wakeupschedule**→**minutesA()**  
**wakeupschedule.get\_minutesA()**

---

**YWakeUpSchedule**

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 interval of each hour scheduled for wake up.

int **get\_minutesB( )**

**Returns :**

an integer corresponding to the minutes in the 30-59 interval of 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\_NEXT\_OCCURENCE\_INVALID.

**wakeupschedule→get\_userData()**

**YWakeUpSchedule**

**wakeupschedule→userData()**

**wakeupschedule.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**wakeupschedule**→**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 watchdog 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 directly 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');
c++	#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 running 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 running state 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 milliseconds

**ms\_duration** pulse duration, in milliseconds

**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` is 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(BadCustomName.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 running state at module power on.

`int get_autoStart()`

**Returns :**

either Y\_AUTOSTART\_OFF or Y\_AUTOSTART\_ON, according to the watchdog running 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 occurred 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 occurred while using the watchdog object

---

**watchdog→get\_friendlyName()****YWatchdog****watchdog→friendlyName()****watchdog.get\_friendlyName()**

---

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

**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 example: `MyCustomName.relay1`)

**Returns :**

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

**watchdog**→**get\_functionDescriptor()**

**YWatchdog**

**watchdog**→**functionDescriptor()**

**watchdog.get\_functionDescriptor()**

---

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

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**watchdog→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 milliseconds

### **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 watchdog 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');
c++	#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(BadCustomName.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 access 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 occurred 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 occurred while using the wireless lan interface object

**wireless→get\_friendlyName()****YWireless****wireless→friendlyName()****wireless.get\_friendlyName()**

---

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

**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 example: `MyCustomName.relay1`)

**Returns :**

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

**wireless→get\_functionDescriptor()**

**YWireless**

**wireless→functionDescriptor()**

**wireless.get\_functionDescriptor()**

---

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

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.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 traffic for instance.

**Parameters :**

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

**Returns :**

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

**wireless→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()**  
**wireless**→**setLogicalName()**  
**wireless.set\_logicalName()**

**YWireless**

---

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

## A

Accelerometer 36  
Access 3  
Activating 4  
adhocNetwork, YWireless 1563  
AnButton 78  
Android 3, 4

## B

Blueprint 16

## C

calibrate, YLightSensor 698  
calibrateFromPoints, YAccelerometer 40  
calibrateFromPoints, YCarbonDioxide 120  
calibrateFromPoints, YCompass 188  
calibrateFromPoints, YCurrent 228  
calibrateFromPoints, YGenericSensor 510  
calibrateFromPoints, YGyro 556  
calibrateFromPoints, YHumidity 632  
calibrateFromPoints, YLightSensor 699  
calibrateFromPoints, YMagnetometer 738  
calibrateFromPoints, YPower 902  
calibrateFromPoints, YPressure 945  
calibrateFromPoints, YQt 1045  
calibrateFromPoints, YSensor 1183  
calibrateFromPoints, YTemperature 1257  
calibrateFromPoints, YTilt 1298  
calibrateFromPoints, YVoc 1337  
calibrateFromPoints, YVoltage 1376  
callbackLogin, YNetwork 823  
cancel3DCalibration, YRefFrame 1111  
CarbonDioxide 116  
CheckLogicalName, YAPI 18  
clear, YDisplayLayer 423  
clearConsole, YDisplayLayer 424  
Clock 1080  
ColorLed 155  
Compass 184  
Compatibility 3  
Configuration 1107  
consoleOut, YDisplayLayer 425  
copyLayerContent, YDisplay 379  
Current 224

## D

Data 294, 304, 316  
DataLogger 263  
delayedPulse, YDigitalIO 335  
delayedPulse, YRelay 1147  
delayedPulse, YWatchdog 1519  
describe, YAccelerometer 41

describe, YAnButton 82  
describe, YCarbonDioxide 121  
describe, YColorLed 158  
describe, YCompass 189  
describe, YCurrent 229  
describe, YDataLogger 266  
describe, YDigitalIO 336  
describe, YDisplay 380  
describe, YDualPower 457  
describe, YFiles 482  
describe, YGenericSensor 511  
describe, YGyro 557  
describe, YHubPort 606  
describe, YHumidity 633  
describe, YLed 670  
describe, YLightSensor 700  
describe, YMagnetometer 739  
describe, YModule 786  
describe, YNetwork 824  
describe, YOsControl 878  
describe, YPower 903  
describe, YPressure 946  
describe, YPwmOutput 984  
describe, YPwmPowerSource 1021  
describe, YQt 1046  
describe, YRealTimeClock 1083  
describe, YRefFrame 1112  
describe, YRelay 1148  
describe, YSensor 1184  
describe, YServo 1222  
describe, YTemperature 1258  
describe, YTilt 1299  
describe, YVoc 1338  
describe, YVoltage 1377  
describe, YVSource 1414  
describe, YWakeUpMonitor 1447  
describe, YWakeUpSchedule 1482  
describe, YWatchdog 1520  
describe, YWireless 1564  
Digital 331  
Display 375  
DisplayLayer 422  
drawBar, YDisplayLayer 426  
drawBitmap, YDisplayLayer 427  
drawCircle, YDisplayLayer 428  
drawDisc, YDisplayLayer 429  
drawImage, YDisplayLayer 430  
drawPixel, YDisplayLayer 431  
drawRect, YDisplayLayer 432  
drawText, YDisplayLayer 433  
dutyCycleMove, YPwmOutput 985

## E

EnableUSBHost, YAPI 19

Error 13  
External 454

## F

fade, YDisplay 381  
Files 479  
FindAccelerometer, YAccelerometer 38  
FindAnButton, YAnButton 80  
FindCarbonDioxide, YCarbonDioxide 118  
FindColorLed, YColorLed 156  
FindCompass, YCompass 186  
FindCurrent, YCurrent 226  
FindDataLogger, YDataLogger 264  
FindDigitalIO, YDigitalIO 333  
FindDisplay, YDisplay 377  
FindDualPower, YDualPower 455  
FindFiles, YFiles 480  
FindGenericSensor, YGenericSensor 508  
FindGyro, YGyro 554  
FindHubPort, YHubPort 604  
FindHumidity, YHumidity 630  
FindLed, YLed 668  
FindLightSensor, YLightSensor 697  
FindMagnetometer, YMagnetometer 737  
FindModule, YModule 785  
FindNetwork, YNetwork 821  
FindOsControl, YOsControl 877  
FindPower, YPower 900  
FindPressure, YPressure 944  
FindPwmOutput, YPwmOutput 983  
FindPwmPowerSource, YPwmPowerSource 1019  
FindQt, YQt 1043  
FindRealTimeClock, YRealTimeClock 1081  
FindRefFrame, YRefFrame 1109  
FindRelay, YRelay 1146  
FindSensor, YSensor 1182  
FindServo, YServo 1221  
FindTemperature, YTemperature 1256  
FindTilt, YTilt 1297  
FindVoc, YVoc 1336  
FindVoltage, YVoltage 1375  
FindVSource, YVSource 1413  
FindWakeUpMonitor, YWakeUpMonitor 1446  
FindWakeUpSchedule, YWakeUpSchedule 1481  
FindWatchdog, YWatchdog 1518  
FindWireless, YWireless 1562  
FirstAccelerometer, YAccelerometer 39  
FirstAnButton, YAnButton 81  
FirstCarbonDioxide, YCarbonDioxide 119  
FirstColorLed, YColorLed 157  
FirstCompass, YCompass 187  
FirstCurrent, YCurrent 227  
FirstDataLogger, YDataLogger 265  
FirstDigitalIO, YDigitalIO 334  
FirstDisplay, YDisplay 378  
FirstDualPower, YDualPower 456  
FirstFiles, YFiles 481  
FirstGenericSensor, YGenericSensor 509

FirstGyro, YGyro 555  
FirstHubPort, YHubPort 605  
FirstHumidity, YHumidity 631  
FirstLed, YLed 669  
FirstLightSensor, YLightSensor 697  
FirstMagnetometer, YMagnetometer 737  
FirstModule, YModule 785  
FirstNetwork, YNetwork 822  
FirstOsControl, YOsControl 877  
FirstPower, YPower 901  
FirstPressure, YPressure 944  
FirstPwmOutput, YPwmOutput 983  
FirstPwmPowerSource, YPwmPowerSource 1020  
FirstQt, YQt 1044  
FirstRealTimeClock, YRealTimeClock 1082  
FirstRefFrame, YRefFrame 1110  
FirstRelay, YRelay 1146  
FirstSensor, YSensor 1182  
FirstServo, YServo 1221  
FirstTemperature, YTemperature 1256  
FirstTilt, YTilt 1297  
FirstVoc, YVoc 1336  
FirstVoltage, YVoltage 1375  
FirstVSource, YVSource 1413  
FirstWakeUpMonitor, YWakeUpMonitor 1446  
FirstWakeUpSchedule, YWakeUpSchedule 1481  
FirstWatchdog, YWatchdog 1518  
FirstWireless, YWireless 1562  
forgetAllDataStreams, YDataLogger 267  
format\_fs, YFiles 483  
Formatted 294  
Frame 1107  
FreeAPI, YAPI 20  
Functions 17

## G

General 17  
GenericSensor 506  
get\_3DCalibrationHint, YRefFrame 1113  
get\_3DCalibrationLogMsg, YRefFrame 1114  
get\_3DCalibrationProgress, YRefFrame 1115  
get\_3DCalibrationStage, YRefFrame 1116  
get\_3DCalibrationStageProgress, YRefFrame 1117  
get\_adminPassword, YNetwork 825  
get\_advertisedValue, YAccelerometer 42  
get\_advertisedValue, YAnButton 83  
get\_advertisedValue, YCarbonDioxide 122  
get\_advertisedValue, YColorLed 159  
get\_advertisedValue, YCompass 190  
get\_advertisedValue, YCurrent 230  
get\_advertisedValue, YDataLogger 268  
get\_advertisedValue, YDigitalIO 337  
get\_advertisedValue, YDisplay 382  
get\_advertisedValue, YDualPower 458  
get\_advertisedValue, YFiles 484  
get\_advertisedValue, YGenericSensor 512  
get\_advertisedValue, YGyro 558

get\_advertisedValue, YHubPort 607  
 get\_advertisedValue, YHumidity 634  
 get\_advertisedValue, YLed 671  
 get\_advertisedValue, YLightSensor 701  
 get\_advertisedValue, YMagnetometer 740  
 get\_advertisedValue, YNetwork 826  
 get\_advertisedValue, YOsControl 879  
 get\_advertisedValue, YPower 904  
 get\_advertisedValue, YPressure 947  
 get\_advertisedValue, YPwmOutput 986  
 get\_advertisedValue, YPwmPowerSource 1022  
 get\_advertisedValue, YQt 1047  
 get\_advertisedValue, YRealTimeClock 1084  
 get\_advertisedValue, YRefFrame 1118  
 get\_advertisedValue, YRelay 1149  
 get\_advertisedValue, YSensor 1185  
 get\_advertisedValue, YServo 1223  
 get\_advertisedValue, YTemperature 1259  
 get\_advertisedValue, YTilt 1300  
 get\_advertisedValue, YVoc 1339  
 get\_advertisedValue, YVoltage 1378  
 get\_advertisedValue, YVSource 1415  
 get\_advertisedValue, YWakeUpMonitor 1448  
 get\_advertisedValue, YWakeUpSchedule 1483  
 get\_advertisedValue, YWatchdog 1521  
 get\_advertisedValue, YWireless 1565  
 get\_analogCalibration, YAnButton 84  
 get\_autoStart, YDataLogger 269  
 get\_autoStart, YWatchdog 1522  
 get\_averageValue, YDataRun 294  
 get\_averageValue, YDataStream 317  
 get\_averageValue, YMeasure 776  
 get\_baudRate, YHubPort 608  
 get\_beacon, YModule 787  
 get\_bearing, YRefFrame 1119  
 get\_bitDirection, YDigitalIO 338  
 get\_bitOpenDrain, YDigitalIO 339  
 get\_bitPolarity, YDigitalIO 340  
 get\_bitState, YDigitalIO 341  
 get\_blinking, YLed 672  
 get\_brightness, YDisplay 383  
 get\_calibratedValue, YAnButton 85  
 get\_calibrationMax, YAnButton 86  
 get\_calibrationMin, YAnButton 87  
 get\_callbackCredentials, YNetwork 827  
 get\_callbackEncoding, YNetwork 828  
 get\_callbackMaxDelay, YNetwork 829  
 get\_callbackMethod, YNetwork 830  
 get\_callbackMinDelay, YNetwork 831  
 get\_callbackUrl, YNetwork 832  
 get\_channel, YWireless 1566  
 get\_columnCount, YDataStream 318  
 get\_columnNames, YDataStream 319  
 get\_cosPhi, YPower 905  
 get\_countdown, YRelay 1150  
 get\_countdown, YWatchdog 1523  
 get\_currentRawValue, YAccelerometer 43  
 get\_currentRawValue, YCarbonDioxide 123  
 get\_currentRawValue, YCompass 191  
 get\_currentRawValue, YCurrent 231  
 get\_currentRawValue, YGenericSensor 513  
 get\_currentRawValue, YGyro 559  
 get\_currentRawValue, YHumidity 635  
 get\_currentRawValue, YLightSensor 702  
 get\_currentRawValue, YMagnetometer 741  
 get\_currentRawValue, YPower 906  
 get\_currentRawValue, YPressure 948  
 get\_currentRawValue, YQt 1048  
 get\_currentRawValue, YSensor 1186  
 get\_currentRawValue, YTemperature 1260  
 get\_currentRawValue, YTilt 1301  
 get\_currentRawValue, YVoc 1340  
 get\_currentRawValue, YVoltage 1379  
 get\_currentRunIndex, YDataLogger 270  
 get\_currentValue, YAccelerometer 44  
 get\_currentValue, YCarbonDioxide 124  
 get\_currentValue, YCompass 192  
 get\_currentValue, YCurrent 232  
 get\_currentValue, YGenericSensor 514  
 get\_currentValue, YGyro 560  
 get\_currentValue, YHumidity 636  
 get\_currentValue, YLightSensor 703  
 get\_currentValue, YMagnetometer 742  
 get\_currentValue, YPower 907  
 get\_currentValue, YPressure 949  
 get\_currentValue, YQt 1049  
 get\_currentValue, YSensor 1187  
 get\_currentValue, YTemperature 1261  
 get\_currentValue, YTilt 1302  
 get\_currentValue, YVoc 1341  
 get\_currentValue, YVoltage 1380  
 get\_data, YDataStream 320  
 get\_dataRows, YDataStream 321  
 get\_dataSamplesIntervalMs, YDataStream 322  
 get\_dataSets, YDataLogger 271  
 get\_dataStreams, YDataLogger 272  
 get\_dateTime, YRealTimeClock 1085  
 get\_detectedWlans, YWireless 1567  
 get\_discoverable, YNetwork 833  
 get\_display, YDisplayLayer 434  
 get\_displayHeight, YDisplay 384  
 get\_displayHeight, YDisplayLayer 435  
 get\_displayLayer, YDisplay 385  
 get\_displayType, YDisplay 386  
 get\_displayWidth, YDisplay 387  
 get\_displayWidth, YDisplayLayer 436  
 get\_duration, YDataRun 295  
 get\_duration, YDataStream 323  
 get\_dutyCycle, YPwmOutput 987  
 get\_dutyCycleAtPowerOn, YPwmOutput 988  
 get\_enabled, YDisplay 388  
 get\_enabled, YHubPort 609  
 get\_enabled, YPwmOutput 989  
 get\_enabled, YServo 1224  
 get\_enabledAtPowerOn, YPwmOutput 990  
 get\_enabledAtPowerOn, YServo 1225  
 get\_endTimeUTC, YDataSet 305  
 get\_endTimeUTC, YMeasure 777

get\_errorMessage, YAccelerometer 45  
get\_errorMessage, YAnButton 88  
get\_errorMessage, YCarbonDioxide 125  
get\_errorMessage, YColorLed 160  
get\_errorMessage, YCompass 193  
get\_errorMessage, YCurrent 233  
get\_errorMessage, YDataLogger 273  
get\_errorMessage, YDigitalIO 342  
get\_errorMessage, YDisplay 389  
get\_errorMessage, YDualPower 459  
get\_errorMessage, YFiles 485  
get\_errorMessage, YGenericSensor 515  
get\_errorMessage, YGyro 561  
get\_errorMessage, YHubPort 610  
get\_errorMessage, YHumidity 637  
get\_errorMessage, YLed 673  
get\_errorMessage, YLightSensor 704  
get\_errorMessage, YMagnetometer 743  
get\_errorMessage, YModule 788  
get\_errorMessage, YNetwork 834  
get\_errorMessage, YOsControl 880  
get\_errorMessage, YPower 908  
get\_errorMessage, YPressure 950  
get\_errorMessage, YPwmOutput 991  
get\_errorMessage, YPwmPowerSource 1023  
get\_errorMessage, YQt 1050  
get\_errorMessage, YRealTimeClock 1086  
get\_errorMessage, YRefFrame 1120  
get\_errorMessage, YRelay 1151  
get\_errorMessage, YSensor 1188  
get\_errorMessage, YServo 1226  
get\_errorMessage, YTemperature 1262  
get\_errorMessage, YTilt 1303  
get\_errorMessage, YVoc 1342  
get\_errorMessage, YVoltage 1381  
get\_errorMessage, YVSource 1416  
get\_errorMessage, YWakeUpMonitor 1449  
get\_errorMessage, YWakeUpSchedule 1484  
get\_errorMessage, YWatchdog 1524  
get\_errorMessage, YWireless 1568  
get\_errorType, YAccelerometer 46  
get\_errorType, YAnButton 89  
get\_errorType, YCarbonDioxide 126  
get\_errorType, YColorLed 161  
get\_errorType, YCompass 194  
get\_errorType, YCurrent 234  
get\_errorType, YDataLogger 274  
get\_errorType, YDigitalIO 343  
get\_errorType, YDisplay 390  
get\_errorType, YDualPower 460  
get\_errorType, YFiles 486  
get\_errorType, YGenericSensor 516  
get\_errorType, YGyro 562  
get\_errorType, YHubPort 611  
get\_errorType, YHumidity 638  
get\_errorType, YLed 674  
get\_errorType, YLightSensor 705  
get\_errorType, YMagnetometer 744  
get\_errorType, YModule 789

get\_errorType, YNetwork 835  
get\_errorType, YOsControl 881  
get\_errorType, YPower 909  
get\_errorType, YPressure 951  
get\_errorType, YPwmOutput 992  
get\_errorType, YPwmPowerSource 1024  
get\_errorType, YQt 1051  
get\_errorType, YRealTimeClock 1087  
get\_errorType, YRefFrame 1121  
get\_errorType, YRelay 1152  
get\_errorType, YSensor 1189  
get\_errorType, YServo 1227  
get\_errorType, YTemperature 1263  
get\_errorType, YTilt 1304  
get\_errorType, YVoc 1343  
get\_errorType, YVoltage 1382  
get\_errorType, YVSource 1417  
get\_errorType, YWakeUpMonitor 1450  
get\_errorType, YWakeUpSchedule 1485  
get\_errorType, YWatchdog 1525  
get\_errorType, YWireless 1569  
get\_extPowerFailure, YVSource 1418  
get\_extVoltage, YDualPower 461  
get\_failure, YVSource 1419  
get\_filesCount, YFiles 487  
get\_firmwareRelease, YModule 790  
get\_freeSpace, YFiles 488  
get\_frequency, YPwmOutput 993  
get\_friendlyName, YAccelerometer 47  
get\_friendlyName, YAnButton 90  
get\_friendlyName, YCarbonDioxide 127  
get\_friendlyName, YColorLed 162  
get\_friendlyName, YCompass 195  
get\_friendlyName, YCurrent 235  
get\_friendlyName, YDataLogger 275  
get\_friendlyName, YDigitalIO 344  
get\_friendlyName, YDisplay 391  
get\_friendlyName, YDualPower 462  
get\_friendlyName, YFiles 489  
get\_friendlyName, YGenericSensor 517  
get\_friendlyName, YGyro 563  
get\_friendlyName, YHubPort 612  
get\_friendlyName, YHumidity 639  
get\_friendlyName, YLed 675  
get\_friendlyName, YLightSensor 706  
get\_friendlyName, YMagnetometer 745  
get\_friendlyName, YNetwork 836  
get\_friendlyName, YOsControl 882  
get\_friendlyName, YPower 910  
get\_friendlyName, YPressure 952  
get\_friendlyName, YPwmOutput 994  
get\_friendlyName, YPwmPowerSource 1025  
get\_friendlyName, YQt 1052  
get\_friendlyName, YRealTimeClock 1088  
get\_friendlyName, YRefFrame 1122  
get\_friendlyName, YRelay 1153  
get\_friendlyName, YSensor 1190  
get\_friendlyName, YServo 1228  
get\_friendlyName, YTemperature 1264

get\_friendlyName, YTilt 1305  
get\_friendlyName, YVoc 1344  
get\_friendlyName, YVoltage 1383  
get\_friendlyName, YVSource 1420  
get\_friendlyName, YWakeUpMonitor 1451  
get\_friendlyName, YWakeUpSchedule 1486  
get\_friendlyName, YWatchdog 1526  
get\_friendlyName, YWireless 1570  
get\_functionDescriptor, YAccelerometer 48  
get\_functionDescriptor, YAnButton 91  
get\_functionDescriptor, YCarbonDioxide 128  
get\_functionDescriptor, YColorLed 163  
get\_functionDescriptor, YCompass 196  
get\_functionDescriptor, YCurrent 236  
get\_functionDescriptor, YDataLogger 276  
get\_functionDescriptor, YDigitalIO 345  
get\_functionDescriptor, YDisplay 392  
get\_functionDescriptor, YDualPower 463  
get\_functionDescriptor, YFiles 490  
get\_functionDescriptor, YGenericSensor 518  
get\_functionDescriptor, YGyro 564  
get\_functionDescriptor, YHubPort 613  
get\_functionDescriptor, YHumidity 640  
get\_functionDescriptor, YLed 676  
get\_functionDescriptor, YLightSensor 707  
get\_functionDescriptor, YMagnetometer 746  
get\_functionDescriptor, YNetwork 837  
get\_functionDescriptor, YOsControl 883  
get\_functionDescriptor, YPower 911  
get\_functionDescriptor, YPressure 953  
get\_functionDescriptor, YPwmOutput 995  
get\_functionDescriptor, YPwmPowerSource 1026  
get\_functionDescriptor, YQt 1053  
get\_functionDescriptor, YRealTimeClock 1089  
get\_functionDescriptor, YRefFrame 1123  
get\_functionDescriptor, YRelay 1154  
get\_functionDescriptor, YSensor 1191  
get\_functionDescriptor, YServo 1229  
get\_functionDescriptor, YTemperature 1265  
get\_functionDescriptor, YTilt 1306  
get\_functionDescriptor, YVoc 1345  
get\_functionDescriptor, YVoltage 1384  
get\_functionDescriptor, YVSource 1421  
get\_functionDescriptor, YWakeUpMonitor 1452  
get\_functionDescriptor, YWakeUpSchedule 1487  
get\_functionDescriptor, YWatchdog 1527  
get\_functionDescriptor, YWireless 1571  
get\_functionId, YAccelerometer 49  
get\_functionId, YAnButton 92  
get\_functionId, YCarbonDioxide 129  
get\_functionId, YColorLed 164  
get\_functionId, YCompass 197  
get\_functionId, YCurrent 237  
get\_functionId, YDataLogger 277  
get\_functionId, YDataSet 306  
get\_functionId, YDigitalIO 346  
get\_functionId, YDisplay 393  
get\_functionId, YDualPower 464  
get\_functionId, YFiles 491

get\_functionId, YGenericSensor 519  
get\_functionId, YGyro 565  
get\_functionId, YHubPort 614  
get\_functionId, YHumidity 641  
get\_functionId, YLed 677  
get\_functionId, YLightSensor 708  
get\_functionId, YMagnetometer 747  
get\_functionId, YNetwork 838  
get\_functionId, YOsControl 884  
get\_functionId, YPower 912  
get\_functionId, YPressure 954  
get\_functionId, YPwmOutput 996  
get\_functionId, YPwmPowerSource 1027  
get\_functionId, YQt 1054  
get\_functionId, YRealTimeClock 1090  
get\_functionId, YRefFrame 1124  
get\_functionId, YRelay 1155  
get\_functionId, YSensor 1192  
get\_functionId, YServo 1230  
get\_functionId, YTemperature 1266  
get\_functionId, YTilt 1307  
get\_functionId, YVoc 1346  
get\_functionId, YVoltage 1385  
get\_functionId, YVSource 1422  
get\_functionId, YWakeUpMonitor 1453  
get\_functionId, YWakeUpSchedule 1488  
get\_functionId, YWatchdog 1528  
get\_functionId, YWireless 1572  
get\_hardwareId, YAccelerometer 50  
get\_hardwareId, YAnButton 93  
get\_hardwareId, YCarbonDioxide 130  
get\_hardwareId, YColorLed 165  
get\_hardwareId, YCompass 198  
get\_hardwareId, YCurrent 238  
get\_hardwareId, YDataLogger 278  
get\_hardwareId, YDataSet 307  
get\_hardwareId, YDigitalIO 347  
get\_hardwareId, YDisplay 394  
get\_hardwareId, YDualPower 465  
get\_hardwareId, YFiles 492  
get\_hardwareId, YGenericSensor 520  
get\_hardwareId, YGyro 566  
get\_hardwareId, YHubPort 615  
get\_hardwareId, YHumidity 642  
get\_hardwareId, YLed 678  
get\_hardwareId, YLightSensor 709  
get\_hardwareId, YMagnetometer 748  
get\_hardwareId, YModule 791  
get\_hardwareId, YNetwork 839  
get\_hardwareId, YOsControl 885  
get\_hardwareId, YPower 913  
get\_hardwareId, YPressure 955  
get\_hardwareId, YPwmOutput 997  
get\_hardwareId, YPwmPowerSource 1028  
get\_hardwareId, YQt 1055  
get\_hardwareId, YRealTimeClock 1091  
get\_hardwareId, YRefFrame 1125  
get\_hardwareId, YRelay 1156  
get\_hardwareId, YSensor 1193

get\_hardwareId, YServo 1231  
get\_hardwareId, YTemperature 1267  
get\_hardwareId, YTilt 1308  
get\_hardwareId, YVoc 1347  
get\_hardwareId, YVoltage 1386  
get\_hardwareId, YVSource 1423  
get\_hardwareId, YWakeUpMonitor 1454  
get\_hardwareId, YWakeUpSchedule 1489  
get\_hardwareId, YWatchdog 1529  
get\_hardwareId, YWireless 1573  
get\_heading, YGyro 567  
get\_highestValue, YAccelerometer 51  
get\_highestValue, YCarbonDioxide 131  
get\_highestValue, YCompass 199  
get\_highestValue, YCurrent 239  
get\_highestValue, YGenericSensor 521  
get\_highestValue, YGyro 568  
get\_highestValue, YHumidity 643  
get\_highestValue, YLightSensor 710  
get\_highestValue, YMagnetometer 749  
get\_highestValue, YPower 914  
get\_highestValue, YPressure 956  
get\_highestValue, YQt 1056  
get\_highestValue, YSensor 1194  
get\_highestValue, YTemperature 1268  
get\_highestValue, YTilt 1309  
get\_highestValue, YVoc 1348  
get\_highestValue, YVoltage 1387  
get\_hours, YWakeUpSchedule 1490  
get\_hslColor, YColorLed 166  
get\_ipAddress, YNetwork 840  
get\_isPressed, YAnButton 94  
get\_lastLogs, YModule 792  
get\_lastTimePressed, YAnButton 95  
get\_lastTimeReleased, YAnButton 96  
get\_layerCount, YDisplay 395  
get\_layerHeight, YDisplay 396  
get\_layerHeight, YDisplayLayer 437  
get\_layerWidth, YDisplay 397  
get\_layerWidth, YDisplayLayer 438  
get\_linkQuality, YWireless 1574  
get\_list, YFiles 493  
get\_logFrequency, YAccelerometer 52  
get\_logFrequency, YCarbonDioxide 132  
get\_logFrequency, YCompass 200  
get\_logFrequency, YCurrent 240  
get\_logFrequency, YGenericSensor 522  
get\_logFrequency, YGyro 569  
get\_logFrequency, YHumidity 644  
get\_logFrequency, YLightSensor 711  
get\_logFrequency, YMagnetometer 750  
get\_logFrequency, YPower 915  
get\_logFrequency, YPressure 957  
get\_logFrequency, YQt 1057  
get\_logFrequency, YSensor 1195  
get\_logFrequency, YTemperature 1269  
get\_logFrequency, YTilt 1310  
get\_logFrequency, YVoc 1349  
get\_logFrequency, YVoltage 1388

get\_logicalName, YAccelerometer 53  
get\_logicalName, YAnButton 97  
get\_logicalName, YCarbonDioxide 133  
get\_logicalName, YColorLed 167  
get\_logicalName, YCompass 201  
get\_logicalName, YCurrent 241  
get\_logicalName, YDataLogger 279  
get\_logicalName, YDigitalIO 348  
get\_logicalName, YDisplay 398  
get\_logicalName, YDualPower 466  
get\_logicalName, YFiles 494  
get\_logicalName, YGenericSensor 523  
get\_logicalName, YGyro 570  
get\_logicalName, YHubPort 616  
get\_logicalName, YHumidity 645  
get\_logicalName, YLed 679  
get\_logicalName, YLightSensor 712  
get\_logicalName, YMagnetometer 751  
get\_logicalName, YModule 793  
get\_logicalName, YNetwork 841  
get\_logicalName, YOsControl 886  
get\_logicalName, YPower 916  
get\_logicalName, YPressure 958  
get\_logicalName, YPwmOutput 998  
get\_logicalName, YPwmPowerSource 1029  
get\_logicalName, YQt 1058  
get\_logicalName, YRealTimeClock 1092  
get\_logicalName, YRefFrame 1126  
get\_logicalName, YRelay 1157  
get\_logicalName, YSensor 1196  
get\_logicalName, YServo 1232  
get\_logicalName, YTemperature 1270  
get\_logicalName, YTilt 1311  
get\_logicalName, YVoc 1350  
get\_logicalName, YVoltage 1389  
get\_logicalName, YVSource 1424  
get\_logicalName, YWakeUpMonitor 1455  
get\_logicalName, YWakeUpSchedule 1491  
get\_logicalName, YWatchdog 1530  
get\_logicalName, YWireless 1575  
get\_lowestValue, YAccelerometer 54  
get\_lowestValue, YCarbonDioxide 134  
get\_lowestValue, YCompass 202  
get\_lowestValue, YCurrent 242  
get\_lowestValue, YGenericSensor 524  
get\_lowestValue, YGyro 571  
get\_lowestValue, YHumidity 646  
get\_lowestValue, YLightSensor 713  
get\_lowestValue, YMagnetometer 752  
get\_lowestValue, YPower 917  
get\_lowestValue, YPressure 959  
get\_lowestValue, YQt 1059  
get\_lowestValue, YSensor 1197  
get\_lowestValue, YTemperature 1271  
get\_lowestValue, YTilt 1312  
get\_lowestValue, YVoc 1351  
get\_lowestValue, YVoltage 1390  
get\_luminosity, YLed 680  
get\_luminosity, YModule 794

get\_macAddress, YNetwork 842  
get\_magneticHeading, YCompass 203  
get\_maxTimeOnStateA, YRelay 1158  
get\_maxTimeOnStateA, YWatchdog 1531  
get\_maxTimeOnStateB, YRelay 1159  
get\_maxTimeOnStateB, YWatchdog 1532  
get\_maxValue, YDataRun 296  
get\_maxValue, YDataStream 324  
get\_maxValue, YMeasure 778  
get\_measureNames, YDataRun 297  
get\_measures, YDataSet 308  
get\_message, YWireless 1576  
get\_meter, YPower 918  
get\_meterTimer, YPower 919  
get\_minutes, YWakeUpSchedule 1492  
get\_minutesA, YWakeUpSchedule 1493  
get\_minutesB, YWakeUpSchedule 1494  
get\_minValue, YDataRun 298  
get\_minValue, YDataStream 325  
get\_minValue, YMeasure 779  
get\_module, YAccelerometer 55  
get\_module, YAnButton 98  
get\_module, YCarbonDioxide 135  
get\_module, YColorLed 168  
get\_module, YCompass 204  
get\_module, YCurrent 243  
get\_module, YDataLogger 280  
get\_module, YDigitalIO 349  
get\_module, YDisplay 399  
get\_module, YDualPower 467  
get\_module, YFiles 495  
get\_module, YGenericSensor 525  
get\_module, YGyro 572  
get\_module, YHubPort 617  
get\_module, YHumidity 647  
get\_module, YLed 681  
get\_module, YLightSensor 714  
get\_module, YMagnetometer 753  
get\_module, YNetwork 843  
get\_module, YOsControl 887  
get\_module, YPower 920  
get\_module, YPressure 960  
get\_module, YPwmOutput 999  
get\_module, YPwmPowerSource 1030  
get\_module, YQt 1060  
get\_module, YRealTimeClock 1093  
get\_module, YRefFrame 1127  
get\_module, YRelay 1160  
get\_module, YSensor 1198  
get\_module, YServo 1233  
get\_module, YTemperature 1272  
get\_module, YTilt 1313  
get\_module, YVoc 1352  
get\_module, YVoltage 1391  
get\_module, YVSource 1425  
get\_module, YWakeUpMonitor 1456  
get\_module, YWakeUpSchedule 1495  
get\_module, YWatchdog 1533  
get\_module, YWireless 1577

get\_monthDays, YWakeUpSchedule 1496  
get\_months, YWakeUpSchedule 1497  
get\_mountOrientation, YRefFrame 1128  
get\_mountPosition, YRefFrame 1129  
get\_neutral, YServo 1234  
get\_nextOccurrence, YWakeUpSchedule 1498  
get\_nextWakeUp, YWakeUpMonitor 1457  
get\_orientation, YDisplay 400  
get\_output, YRelay 1161  
get\_output, YWatchdog 1534  
get\_outputVoltage, YDigitalIO 350  
get\_overCurrent, YVSource 1426  
get\_overHeat, YVSource 1427  
get\_overLoad, YVSource 1428  
get\_period, YPwmOutput 1000  
get\_persistentSettings, YModule 795  
get\_pitch, YGyro 573  
get\_poeCurrent, YNetwork 844  
get\_portDirection, YDigitalIO 351  
get\_portOpenDrain, YDigitalIO 352  
get\_portPolarity, YDigitalIO 353  
get\_portSize, YDigitalIO 354  
get\_portState, YDigitalIO 355  
get\_portState, YHubPort 618  
get\_position, YServo 1235  
get\_positionAtPowerOn, YServo 1236  
get\_power, YLed 682  
get\_powerControl, YDualPower 468  
get\_powerDuration, YWakeUpMonitor 1458  
get\_powerMode, YPwmPowerSource 1031  
get\_powerState, YDualPower 469  
get\_preview, YDataSet 309  
get\_primaryDNS, YNetwork 845  
get\_productId, YModule 796  
get\_productName, YModule 797  
get\_productRelease, YModule 798  
get\_progress, YDataSet 310  
get\_pulseCounter, YAnButton 99  
get\_pulseDuration, YPwmOutput 1001  
get\_pulseTimer, YAnButton 100  
get\_pulseTimer, YRelay 1162  
get\_pulseTimer, YWatchdog 1535  
get\_quaternionW, YGyro 574  
get\_quaternionX, YGyro 575  
get\_quaternionY, YGyro 576  
get\_quaternionZ, YGyro 577  
get\_range, YServo 1237  
get\_rawValue, YAnButton 101  
get\_readiness, YNetwork 846  
get\_rebootCountdown, YModule 799  
get\_recordedData, YAccelerometer 56  
get\_recordedData, YCarbonDioxide 136  
get\_recordedData, YCompass 205  
get\_recordedData, YCurrent 244  
get\_recordedData, YGenericSensor 526  
get\_recordedData, YGyro 578  
get\_recordedData, YHumidity 648  
get\_recordedData, YLightSensor 715  
get\_recordedData, YMagnetometer 754

get\_recordedData, YPower 921  
get\_recordedData, YPressure 961  
get\_recordedData, YQt 1061  
get\_recordedData, YSensor 1199  
get\_recordedData, YTemperature 1273  
get\_recordedData, YTilt 1314  
get\_recordedData, YVoc 1353  
get\_recordedData, YVoltage 1392  
get\_recording, YDataLogger 281  
get\_regulationFailure, YVSource 1429  
get\_reportFrequency, YAccelerometer 57  
get\_reportFrequency, YCarbonDioxide 137  
get\_reportFrequency, YCompass 206  
get\_reportFrequency, YCurrent 245  
get\_reportFrequency, YGenericSensor 527  
get\_reportFrequency, YGyro 579  
get\_reportFrequency, YHumidity 649  
get\_reportFrequency, YLightSensor 716  
get\_reportFrequency, YMagnetometer 755  
get\_reportFrequency, YPower 922  
get\_reportFrequency, YPressure 962  
get\_reportFrequency, YQt 1062  
get\_reportFrequency, YSensor 1200  
get\_reportFrequency, YTemperature 1274  
get\_reportFrequency, YTilt 1315  
get\_reportFrequency, YVoc 1354  
get\_reportFrequency, YVoltage 1393  
get\_resolution, YAccelerometer 58  
get\_resolution, YCarbonDioxide 138  
get\_resolution, YCompass 207  
get\_resolution, YCurrent 246  
get\_resolution, YGenericSensor 528  
get\_resolution, YGyro 580  
get\_resolution, YHumidity 650  
get\_resolution, YLightSensor 717  
get\_resolution, YMagnetometer 756  
get\_resolution, YPower 923  
get\_resolution, YPressure 963  
get\_resolution, YQt 1063  
get\_resolution, YSensor 1201  
get\_resolution, YTemperature 1275  
get\_resolution, YTilt 1316  
get\_resolution, YVoc 1355  
get\_resolution, YVoltage 1394  
get\_rgbColor, YColorLed 169  
get\_rgbColorAtPowerOn, YColorLed 170  
get\_roll, YGyro 581  
get\_router, YNetwork 847  
get\_rowCount, YDataStream 326  
get\_runIndex, YDataStream 327  
get\_running, YWatchdog 1536  
get\_secondaryDNS, YNetwork 848  
get\_security, YWireless 1578  
get\_sensitivity, YAnButton 102  
get\_sensorType, YTemperature 1276  
get\_serialNumber, YModule 800  
get\_shutdownCountdown, YOsControl 888  
get\_signalRange, YGenericSensor 529  
get\_signalUnit, YGenericSensor 530  
get\_signalValue, YGenericSensor 531  
get\_sleepCountdown, YWakeUpMonitor 1459  
get\_ssid, YWireless 1579  
get\_startTime, YDataStream 328  
get\_startTimeUTC, YDataRun 299  
get\_startTimeUTC, YDataSet 311  
get\_startTimeUTC, YDataStream 329  
get\_startTimeUTC, YMeasure 780  
get\_startupSeq, YDisplay 401  
get\_state, YRelay 1163  
get\_state, YWatchdog 1537  
get\_stateAtPowerOn, YRelay 1164  
get\_stateAtPowerOn, YWatchdog 1538  
get\_subnetMask, YNetwork 849  
get\_summary, YDataSet 312  
get\_timeSet, YRealTimeClock 1094  
get\_timeUTC, YDataLogger 282  
get\_triggerDelay, YWatchdog 1539  
get\_triggerDuration, YWatchdog 1540  
get\_unit, YAccelerometer 59  
get\_unit, YCarbonDioxide 139  
get\_unit, YCompass 208  
get\_unit, YCurrent 247  
get\_unit, YDataSet 313  
get\_unit, YGenericSensor 532  
get\_unit, YGyro 582  
get\_unit, YHumidity 651  
get\_unit, YLightSensor 718  
get\_unit, YMagnetometer 757  
get\_unit, YPower 924  
get\_unit, YPressure 964  
get\_unit, YQt 1064  
get\_unit, YSensor 1202  
get\_unit, YTemperature 1277  
get\_unit, YTilt 1317  
get\_unit, YVoc 1356  
get\_unit, YVoltage 1395  
get\_unit, YVSource 1430  
get\_unixTime, YRealTimeClock 1095  
get\_upTime, YModule 801  
get\_usbBandwidth, YModule 802  
get\_usbCurrent, YModule 803  
get\_userData, YAccelerometer 60  
get\_userData, YAnButton 103  
get\_userData, YCarbonDioxide 140  
get\_userData, YColorLed 171  
get\_userData, YCompass 209  
get\_userData, YCurrent 248  
get\_userData, YDataLogger 283  
get\_userData, YDigitalIO 356  
get\_userData, YDisplay 402  
get\_userData, YDualPower 470  
get\_userData, YFiles 496  
get\_userData, YGenericSensor 533  
get\_userData, YGyro 583  
get\_userData, YHubPort 619  
get\_userData, YHumidity 652  
get\_userData, YLed 683  
get\_userData, YLightSensor 719



- get\_userdata, YMagnetometer 758
- get\_userdata, YModule 804
- get\_userdata, YNetwork 850
- get\_userdata, YOsControl 889
- get\_userdata, YPower 925
- get\_userdata, YPressure 965
- get\_userdata, YPwmOutput 1002
- get\_userdata, YPwmPowerSource 1032
- get\_userdata, YQt 1065
- get\_userdata, YRealTimeClock 1096
- get\_userdata, YRefFrame 1130
- get\_userdata, YRelay 1165
- get\_userdata, YSensor 1203
- get\_userdata, YServo 1238
- get\_userdata, YTemperature 1278
- get\_userdata, YTilt 1318
- get\_userdata, YVoc 1357
- get\_userdata, YVoltage 1396
- get\_userdata, YVSource 1431
- get\_userdata, YWakeUpMonitor 1460
- get\_userdata, YWakeUpSchedule 1499
- get\_userdata, YWatchdog 1541
- get\_userdata, YWireless 1580
- get\_userPassword, YNetwork 851
- get\_utcOffset, YRealTimeClock 1097
- get\_valueCount, YDataRun 300
- get\_valueInterval, YDataRun 301
- get\_valueRange, YGenericSensor 534
- get\_voltage, YVSource 1432
- get\_wakeUpReason, YWakeUpMonitor 1461
- get\_wakeUpState, YWakeUpMonitor 1462
- get\_weekDays, YWakeUpSchedule 1500
- get\_wwwWatchdogDelay, YNetwork 852
- get\_xValue, YAccelerometer 61
- get\_xValue, YGyro 584
- get\_xValue, YMagnetometer 759
- get\_yValue, YAccelerometer 62
- get\_yValue, YGyro 585
- get\_yValue, YMagnetometer 760
- get\_zValue, YAccelerometer 63
- get\_zValue, YGyro 586
- get\_zValue, YMagnetometer 761
- GetAPIVersion, YAPI 21
- GetTickCount, YAPI 22
- Gyroscope 552

## H

- HandleEvents, YAPI 23
- hide, YDisplayLayer 439
- hslMove, YColorLed 172
- Humidity 628

## I

- InitAPI, YAPI 24
- Interface 36, 78, 116, 155, 184, 224, 263, 331, 375, 422, 454, 479, 506, 552, 603, 628, 667, 694, 734, 782, 818, 898, 941, 980, 1018, 1041, 1080, 1143, 1179, 1218, 1253, 1294, 1333,

- 1372, 1411, 1443, 1478, 1515, 1560
- Introduction 1
- isOnline, YAccelerometer 64
- isOnline, YAnButton 104
- isOnline, YCarbonDioxide 141
- isOnline, YColorLed 173
- isOnline, YCompass 210
- isOnline, YCurrent 249
- isOnline, YDataLogger 284
- isOnline, YDigitalIO 357
- isOnline, YDisplay 403
- isOnline, YDualPower 471
- isOnline, YFiles 497
- isOnline, YGenericSensor 535
- isOnline, YGyro 587
- isOnline, YHubPort 620
- isOnline, YHumidity 653
- isOnline, YLed 684
- isOnline, YLightSensor 720
- isOnline, YMagnetometer 762
- isOnline, YModule 805
- isOnline, YNetwork 853
- isOnline, YOsControl 890
- isOnline, YPower 926
- isOnline, YPressure 966
- isOnline, YPwmOutput 1003
- isOnline, YPwmPowerSource 1033
- isOnline, YQt 1066
- isOnline, YRealTimeClock 1098
- isOnline, YRefFrame 1131
- isOnline, YRelay 1166
- isOnline, YSensor 1204
- isOnline, YServo 1239
- isOnline, YTemperature 1279
- isOnline, YTilt 1319
- isOnline, YVoc 1358
- isOnline, YVoltage 1397
- isOnline, YVSource 1433
- isOnline, YWakeUpMonitor 1463
- isOnline, YWakeUpSchedule 1501
- isOnline, YWatchdog 1542
- isOnline, YWireless 1581

## J

- joinNetwork, YWireless 1582

## L

- LightSensor 694
- lineTo, YDisplayLayer 440
- load, YAccelerometer 65
- load, YAnButton 105
- load, YCarbonDioxide 142
- load, YColorLed 174
- load, YCompass 211
- load, YCurrent 250
- load, YDataLogger 285
- load, YDigitalIO 358
- load, YDisplay 404

- load, YDualPower 472
- load, YFiles 498
- load, YGenericSensor 536
- load, YGyro 588
- load, YHubPort 621
- load, YHumidity 654
- load, YLed 685
- load, YLightSensor 721
- load, YMagnetometer 763
- load, YModule 806
- load, YNetwork 854
- load, YOsControl 891
- load, YPower 927
- load, YPressure 967
- load, YPwmOutput 1004
- load, YPwmPowerSource 1034
- load, YQt 1067
- load, YRealTimeClock 1099
- load, YRefFrame 1132
- load, YRelay 1167
- load, YSensor 1205
- load, YServo 1240
- load, YTemperature 1280
- load, YTilt 1320
- load, YVoc 1359
- load, YVoltage 1398
- load, YVSource 1434
- load, YWakeUpMonitor 1464
- load, YWakeUpSchedule 1502
- load, YWatchdog 1543
- load, YWireless 1583
- loadCalibrationPoints, YAccelerometer 66
- loadCalibrationPoints, YCarbonDioxide 143
- loadCalibrationPoints, YCompass 212
- loadCalibrationPoints, YCurrent 251
- loadCalibrationPoints, YGenericSensor 537
- loadCalibrationPoints, YGyro 589
- loadCalibrationPoints, YHumidity 655
- loadCalibrationPoints, YLightSensor 722
- loadCalibrationPoints, YMagnetometer 764
- loadCalibrationPoints, YPower 928
- loadCalibrationPoints, YPressure 968
- loadCalibrationPoints, YQt 1068
- loadCalibrationPoints, YSensor 1206
- loadCalibrationPoints, YTemperature 1281
- loadCalibrationPoints, YTilt 1321
- loadCalibrationPoints, YVoc 1360
- loadCalibrationPoints, YVoltage 1399
- loadMore, YDataSet 314

## M

- Magnetometer 734
- Measured 776
- Module 8, 782
- more3DCalibration, YRefFrame 1133
- move, YServo 1241
- moveTo, YDisplayLayer 441

## N

- Native 3
- Network 818
- newSequence, YDisplay 405
- nextAccelerometer, YAccelerometer 67
- nextAnButton, YAnButton 106
- nextCarbonDioxide, YCarbonDioxide 144
- nextColorLed, YColorLed 175
- nextCompass, YCompass 213
- nextCurrent, YCurrent 252
- nextDataLogger, YDataLogger 286
- nextDigitalIO, YDigitalIO 359
- nextDisplay, YDisplay 406
- nextDualPower, YDualPower 473
- nextFiles, YFiles 499
- nextGenericSensor, YGenericSensor 538
- nextGyro, YGyro 590
- nextHubPort, YHubPort 622
- nextHumidity, YHumidity 656
- nextLed, YLed 686
- nextLightSensor, YLightSensor 723
- nextMagnetometer, YMagnetometer 765
- nextModule, YModule 807
- nextNetwork, YNetwork 855
- nextOsControl, YOsControl 892
- nextPower, YPower 929
- nextPressure, YPressure 969
- nextPwmOutput, YPwmOutput 1005
- nextPwmPowerSource, YPwmPowerSource 1035
- nextQt, YQt 1069
- nextRealTimeClock, YRealTimeClock 1100
- nextRefFrame, YRefFrame 1134
- nextRelay, YRelay 1168
- nextSensor, YSensor 1207
- nextServo, YServo 1242
- nextTemperature, YTemperature 1282
- nextTilt, YTilt 1322
- nextVoc, YVoc 1361
- nextVoltage, YVoltage 1400
- nextVSource, YVSource 1435
- nextWakeUpMonitor, YWakeUpMonitor 1465
- nextWakeUpSchedule, YWakeUpSchedule 1503
- nextWatchdog, YWatchdog 1544
- nextWireless, YWireless 1584

## O

- Object 422

## P

- pauseSequence, YDisplay 407
- ping, YNetwork 856
- playSequence, YDisplay 408
- Port 4, 603
- Power 454, 898
- PreregisterHub, YAPI 25

Pressure 941  
pulse, YDigitalIO 360  
pulse, YRelay 1169  
pulse, YVSource 1436  
pulse, YWatchdog 1545  
pulseDurationMove, YPwmOutput 1006  
PwmPowerSource 1018

## Q

Quaternion 1041

## R

Real 1080  
reboot, YModule 808  
Recorded 304  
Reference 16, 1107  
registerAnglesCallback, YGyro 591  
RegisterDeviceArrivalCallback, YAPI 26  
RegisterDeviceRemovalCallback, YAPI 27  
RegisterHub, YAPI 28  
RegisterHubDiscoveryCallback, YAPI 29  
RegisterLogFunction, YAPI 30  
registerQuaternionCallback, YGyro 592  
registerTimedReportCallback, YAccelerometer 68  
registerTimedReportCallback, YCarbonDioxide 145  
registerTimedReportCallback, YCompass 214  
registerTimedReportCallback, YCurrent 253  
registerTimedReportCallback, YGenericSensor 539  
registerTimedReportCallback, YGyro 593  
registerTimedReportCallback, YHumidity 657  
registerTimedReportCallback, YLightSensor 724  
registerTimedReportCallback, YMagnetometer 766  
registerTimedReportCallback, YPower 930  
registerTimedReportCallback, YPressure 970  
registerTimedReportCallback, YQt 1070  
registerTimedReportCallback, YSensor 1208  
registerTimedReportCallback, YTemperature 1283  
registerTimedReportCallback, YTilt 1323  
registerTimedReportCallback, YVoc 1362  
registerTimedReportCallback, YVoltage 1401  
registerValueCallback, YAccelerometer 69  
registerValueCallback, YAnButton 107  
registerValueCallback, YCarbonDioxide 146  
registerValueCallback, YColorLed 176  
registerValueCallback, YCompass 215  
registerValueCallback, YCurrent 254  
registerValueCallback, YDataLogger 287  
registerValueCallback, YDigitalIO 361  
registerValueCallback, YDisplay 409  
registerValueCallback, YDualPower 474  
registerValueCallback, YFiles 500  
registerValueCallback, YGenericSensor 540  
registerValueCallback, YGyro 594

registerValueCallback, YHubPort 623  
registerValueCallback, YHumidity 658  
registerValueCallback, YLed 687  
registerValueCallback, YLightSensor 725  
registerValueCallback, YMagnetometer 767  
registerValueCallback, YNetwork 857  
registerValueCallback, YOsControl 893  
registerValueCallback, YPower 931  
registerValueCallback, YPressure 971  
registerValueCallback, YPwmOutput 1007  
registerValueCallback, YPwmPowerSource 1036  
registerValueCallback, YQt 1071  
registerValueCallback, YRealTimeClock 1101  
registerValueCallback, YRefFrame 1135  
registerValueCallback, YRelay 1170  
registerValueCallback, YSensor 1209  
registerValueCallback, YServo 1243  
registerValueCallback, YTemperature 1284  
registerValueCallback, YTilt 1324  
registerValueCallback, YVoc 1363  
registerValueCallback, YVoltage 1402  
registerValueCallback, YVSource 1437  
registerValueCallback, YWakeUpMonitor 1466  
registerValueCallback, YWakeUpSchedule 1504  
registerValueCallback, YWatchdog 1546  
registerValueCallback, YWireless 1585  
Relay 1143  
remove, YFiles 501  
reset, YDisplayLayer 442  
reset, YPower 932  
resetAll, YDisplay 410  
resetCounter, YAnButton 108  
resetSleepCountDown, YWakeUpMonitor 1467  
resetWatchdog, YWatchdog 1547  
revertFromFlash, YModule 809  
rgbMove, YColorLed 177

## S

save3DCalibration, YRefFrame 1136  
saveSequence, YDisplay 411  
saveToFlash, YModule 810  
selectColorPen, YDisplayLayer 443  
selectEraser, YDisplayLayer 444  
selectFont, YDisplayLayer 445  
selectGrayPen, YDisplayLayer 446  
Sensor 1179  
Sequence 294, 304, 316  
Servo 1218  
set\_adminPassword, YNetwork 858  
set\_analogCalibration, YAnButton 109  
set\_autoStart, YDataLogger 288  
set\_autoStart, YWatchdog 1548  
set\_beacon, YModule 811  
set\_bearing, YRefFrame 1137  
set\_bitDirection, YDigitalIO 362  
set\_bitOpenDrain, YDigitalIO 363  
set\_bitPolarity, YDigitalIO 364  
set\_bitState, YDigitalIO 365  
set\_blinking, YLed 688

set\_brightness, YDisplay 412  
 set\_calibrationMax, YAnButton 110  
 set\_calibrationMin, YAnButton 111  
 set\_callbackCredentials, YNetwork 859  
 set\_callbackEncoding, YNetwork 860  
 set\_callbackMaxDelay, YNetwork 861  
 set\_callbackMethod, YNetwork 862  
 set\_callbackMinDelay, YNetwork 863  
 set\_callbackUrl, YNetwork 864  
 set\_discoverable, YNetwork 865  
 set\_dutyCycle, YPwmOutput 1008  
 set\_dutyCycleAtPowerOn, YPwmOutput 1009  
 set\_enabled, YDisplay 413  
 set\_enabled, YHubPort 624  
 set\_enabled, YPwmOutput 1010  
 set\_enabled, YServo 1244  
 set\_enabledAtPowerOn, YPwmOutput 1011  
 set\_enabledAtPowerOn, YServo 1245  
 set\_frequency, YPwmOutput 1012  
 set\_highestValue, YAccelerometer 70  
 set\_highestValue, YCarbonDioxide 147  
 set\_highestValue, YCompass 216  
 set\_highestValue, YCurrent 255  
 set\_highestValue, YGenericSensor 541  
 set\_highestValue, YGyro 595  
 set\_highestValue, YHumidity 659  
 set\_highestValue, YLightSensor 726  
 set\_highestValue, YMagnetometer 768  
 set\_highestValue, YPower 933  
 set\_highestValue, YPressure 972  
 set\_highestValue, YQt 1072  
 set\_highestValue, YSensor 1210  
 set\_highestValue, YTemperature 1285  
 set\_highestValue, YTilt 1325  
 set\_highestValue, YVoc 1364  
 set\_highestValue, YVoltage 1403  
 set\_hours, YWakeUpSchedule 1505  
 set\_hslColor, YColorLed 178  
 set\_logFrequency, YAccelerometer 71  
 set\_logFrequency, YCarbonDioxide 148  
 set\_logFrequency, YCompass 217  
 set\_logFrequency, YCurrent 256  
 set\_logFrequency, YGenericSensor 542  
 set\_logFrequency, YGyro 596  
 set\_logFrequency, YHumidity 660  
 set\_logFrequency, YLightSensor 727  
 set\_logFrequency, YMagnetometer 769  
 set\_logFrequency, YPower 934  
 set\_logFrequency, YPressure 973  
 set\_logFrequency, YQt 1073  
 set\_logFrequency, YSensor 1211  
 set\_logFrequency, YTemperature 1286  
 set\_logFrequency, YTilt 1326  
 set\_logFrequency, YVoc 1365  
 set\_logFrequency, YVoltage 1404  
 set\_logicalName, YAccelerometer 72  
 set\_logicalName, YAnButton 112  
 set\_logicalName, YCarbonDioxide 149  
 set\_logicalName, YColorLed 179  
 set\_logicalName, YCompass 218  
 set\_logicalName, YCurrent 257  
 set\_logicalName, YDataLogger 289  
 set\_logicalName, YDigitalIO 366  
 set\_logicalName, YDisplay 414  
 set\_logicalName, YDualPower 475  
 set\_logicalName, YFiles 502  
 set\_logicalName, YGenericSensor 543  
 set\_logicalName, YGyro 597  
 set\_logicalName, YHubPort 625  
 set\_logicalName, YHumidity 661  
 set\_logicalName, YLed 689  
 set\_logicalName, YLightSensor 728  
 set\_logicalName, YMagnetometer 770  
 set\_logicalName, YModule 812  
 set\_logicalName, YNetwork 866  
 set\_logicalName, YOsControl 894  
 set\_logicalName, YPower 935  
 set\_logicalName, YPressure 974  
 set\_logicalName, YPwmOutput 1013  
 set\_logicalName, YPwmPowerSource 1037  
 set\_logicalName, YQt 1074  
 set\_logicalName, YRealTimeClock 1102  
 set\_logicalName, YRefFrame 1138  
 set\_logicalName, YRelay 1171  
 set\_logicalName, YSensor 1212  
 set\_logicalName, YServo 1246  
 set\_logicalName, YTemperature 1287  
 set\_logicalName, YTilt 1327  
 set\_logicalName, YVoc 1366  
 set\_logicalName, YVoltage 1405  
 set\_logicalName, YVSource 1438  
 set\_logicalName, YWakeUpMonitor 1468  
 set\_logicalName, YWakeUpSchedule 1506  
 set\_logicalName, YWatchdog 1549  
 set\_logicalName, YWireless 1586  
 set\_lowestValue, YAccelerometer 73  
 set\_lowestValue, YCarbonDioxide 150  
 set\_lowestValue, YCompass 219  
 set\_lowestValue, YCurrent 258  
 set\_lowestValue, YGenericSensor 544  
 set\_lowestValue, YGyro 598  
 set\_lowestValue, YHumidity 662  
 set\_lowestValue, YLightSensor 729  
 set\_lowestValue, YMagnetometer 771  
 set\_lowestValue, YPower 936  
 set\_lowestValue, YPressure 975  
 set\_lowestValue, YQt 1075  
 set\_lowestValue, YSensor 1213  
 set\_lowestValue, YTemperature 1288  
 set\_lowestValue, YTilt 1328  
 set\_lowestValue, YVoc 1367  
 set\_lowestValue, YVoltage 1406  
 set\_luminosity, YLed 690  
 set\_luminosity, YModule 813  
 set\_maxTimeOnStateA, YRelay 1172  
 set\_maxTimeOnStateA, YWatchdog 1550  
 set\_maxTimeOnStateB, YRelay 1173  
 set\_maxTimeOnStateB, YWatchdog 1551

set\_minutes, YWakeUpSchedule 1507  
set\_minutesA, YWakeUpSchedule 1508  
set\_minutesB, YWakeUpSchedule 1509  
set\_monthDays, YWakeUpSchedule 1510  
set\_months, YWakeUpSchedule 1511  
set\_mountPosition, YRefFrame 1139  
set\_neutral, YServo 1247  
set\_nextWakeUp, YWakeUpMonitor 1469  
set\_orientation, YDisplay 415  
set\_output, YRelay 1174  
set\_output, YWatchdog 1552  
set\_outputVoltage, YDigitalIO 367  
set\_period, YPwmOutput 1014  
set\_portDirection, YDigitalIO 368  
set\_portOpenDrain, YDigitalIO 369  
set\_portPolarity, YDigitalIO 370  
set\_portState, YDigitalIO 371  
set\_position, YServo 1248  
set\_positionAtPowerOn, YServo 1249  
set\_power, YLed 691  
set\_powerControl, YDualPower 476  
set\_powerDuration, YWakeUpMonitor 1470  
set\_powerMode, YPwmPowerSource 1038  
set\_primaryDNS, YNetwork 867  
set\_pulseDuration, YPwmOutput 1015  
set\_range, YServo 1250  
set\_recording, YDataLogger 290  
set\_reportFrequency, YAccelerometer 74  
set\_reportFrequency, YCarbonDioxide 151  
set\_reportFrequency, YCompass 220  
set\_reportFrequency, YCurrent 259  
set\_reportFrequency, YGenericSensor 545  
set\_reportFrequency, YGyro 599  
set\_reportFrequency, YHumidity 663  
set\_reportFrequency, YLightSensor 730  
set\_reportFrequency, YMagnetometer 772  
set\_reportFrequency, YPower 937  
set\_reportFrequency, YPressure 976  
set\_reportFrequency, YQt 1076  
set\_reportFrequency, YSensor 1214  
set\_reportFrequency, YTemperature 1289  
set\_reportFrequency, YTilt 1329  
set\_reportFrequency, YVoc 1368  
set\_reportFrequency, YVoltage 1407  
set\_resolution, YAccelerometer 75  
set\_resolution, YCarbonDioxide 152  
set\_resolution, YCompass 221  
set\_resolution, YCurrent 260  
set\_resolution, YGenericSensor 546  
set\_resolution, YGyro 600  
set\_resolution, YHumidity 664  
set\_resolution, YLightSensor 731  
set\_resolution, YMagnetometer 773  
set\_resolution, YPower 938  
set\_resolution, YPressure 977  
set\_resolution, YQt 1077  
set\_resolution, YSensor 1215  
set\_resolution, YTemperature 1290  
set\_resolution, YTilt 1330

set\_resolution, YVoc 1369  
set\_resolution, YVoltage 1408  
set\_rgbColor, YColorLed 180  
set\_rgbColorAtPowerOn, YColorLed 181  
set\_running, YWatchdog 1553  
set\_secondaryDNS, YNetwork 868  
set\_sensitivity, YAnButton 113  
set\_sensorType, YTemperature 1291  
set\_signalRange, YGenericSensor 547  
set\_sleepCountdown, YWakeUpMonitor 1471  
set\_startupSeq, YDisplay 416  
set\_state, YRelay 1175  
set\_state, YWatchdog 1554  
set\_stateAtPowerOn, YRelay 1176  
set\_stateAtPowerOn, YWatchdog 1555  
set\_timeUTC, YDataLogger 291  
set\_triggerDelay, YWatchdog 1556  
set\_triggerDuration, YWatchdog 1557  
set\_unit, YGenericSensor 548  
set\_unixTime, YRealTimeClock 1103  
set\_usbBandwidth, YModule 814  
set\_userData, YAccelerometer 76  
set\_userData, YAnButton 114  
set\_userData, YCarbonDioxide 153  
set\_userData, YColorLed 182  
set\_userData, YCompass 222  
set\_userData, YCurrent 261  
set\_userData, YDataLogger 292  
set\_userData, YDigitalIO 372  
set\_userData, YDisplay 417  
set\_userData, YDualPower 477  
set\_userData, YFiles 503  
set\_userData, YGenericSensor 549  
set\_userData, YGyro 601  
set\_userData, YHubPort 626  
set\_userData, YHumidity 665  
set\_userData, YLed 692  
set\_userData, YLightSensor 732  
set\_userData, YMagnetometer 774  
set\_userData, YModule 815  
set\_userData, YNetwork 869  
set\_userData, YOsControl 895  
set\_userData, YPower 939  
set\_userData, YPressure 978  
set\_userData, YPwmOutput 1016  
set\_userData, YPwmPowerSource 1039  
set\_userData, YQt 1078  
set\_userData, YRealTimeClock 1104  
set\_userData, YRefFrame 1140  
set\_userData, YRelay 1177  
set\_userData, YSensor 1216  
set\_userData, YServo 1251  
set\_userData, YTemperature 1292  
set\_userData, YTilt 1331  
set\_userData, YVoc 1370  
set\_userData, YVoltage 1409  
set\_userData, YVSource 1439  
set\_userData, YWakeUpMonitor 1472  
set\_userData, YWakeUpSchedule 1512

set\_userdata, YWatchdog 1558  
set\_userdata, YWireless 1587  
set\_userPassword, YNetwork 870  
set\_utcOffset, YRealTimeClock 1105  
set\_valueInterval, YDataRun 302  
set\_valueRange, YGenericSensor 550  
set\_voltage, YVSource 1440  
set\_weekDays, YWakeUpSchedule 1513  
set\_wwwWatchdogDelay, YNetwork 871  
setAntialiasingMode, YDisplayLayer 447  
setConsoleBackground, YDisplayLayer 448  
setConsoleMargins, YDisplayLayer 449  
setConsoleWordWrap, YDisplayLayer 450  
setLayerPosition, YDisplayLayer 451  
shutdown, YOsControl 896  
Sleep, YAPI 31  
sleep, YWakeUpMonitor 1473  
sleepFor, YWakeUpMonitor 1474  
sleepUntil, YWakeUpMonitor 1475  
Source 1411  
start3DCalibration, YRefFrame 1141  
stopSequence, YDisplay 418  
Supply 454  
swapLayerContent, YDisplay 419

## T

Temperature 1253  
Tilt 1294  
Time 1080  
toggle\_bitState, YDigitalIO 373  
triggerFirmwareUpdate, YModule 816  
TriggerHubDiscovery, YAPI 32

## U

Unformatted 316  
unhide, YDisplayLayer 452  
UnregisterHub, YAPI 33  
UpdateDeviceList, YAPI 34  
upload, YDisplay 420  
upload, YFiles 504  
useDHCP, YNetwork 872  
useStaticIP, YNetwork 873

## V

Value 776  
VirtualHub 3  
Voltage 1372, 1411  
voltageMove, YVSource 1441

## W

wakeUp, YWakeUpMonitor 1476  
WakeUpMonitor 1443  
WakeUpSchedule 1478  
Watchdog 1515  
Wireless 1560

## Y

YAccelerometer 38-76  
YAnButton 80-114  
YAPI 18-34  
YCarbonDioxide 118-153  
yCheckLogicalName 18  
YColorLed 156-182  
YCompass 186-222  
YCurrent 226-261  
YDataLogger 264-292  
YDataRun 294-302  
YDataSet 305-314  
YDataStream 317-329  
YDigitalIO 333-373  
YDisplay 377-420  
YDisplayLayer 423-452  
YDualPower 455-477  
yEnableUSBHost 19  
YFiles 480-504  
yFindAccelerometer 38  
yFindAnButton 80  
yFindCarbonDioxide 118  
yFindColorLed 156  
yFindCompass 186  
yFindCurrent 226  
yFindDataLogger 264  
yFindDigitalIO 333  
yFindDisplay 377  
yFindDualPower 455  
yFindFiles 480  
yFindGenericSensor 508  
yFindGyro 554  
yFindHubPort 604  
yFindHumidity 630  
yFindLed 668  
yFindLightSensor 696  
yFindMagnetometer 736  
yFindModule 784  
yFindNetwork 821  
yFindOsControl 876  
yFindPower 900  
yFindPressure 943  
yFindPwmOutput 982  
yFindPwmPowerSource 1019  
yFindQt 1043  
yFindRealTimeClock 1081  
yFindRefFrame 1109  
yFindRelay 1145  
yFindSensor 1181  
yFindServo 1220  
yFindTemperature 1255  
yFindTilt 1296  
yFindVoc 1335  
yFindVoltage 1374  
yFindVSource 1412  
yFindWakeUpMonitor 1445  
yFindWakeUpSchedule 1480  
yFindWatchdog 1517

yFindWireless 1561  
yFirstAccelerometer 39  
yFirstAnButton 81  
yFirstCarbonDioxide 119  
yFirstColorLed 157  
yFirstCompass 187  
yFirstCurrent 227  
yFirstDataLogger 265  
yFirstDigitalIO 334  
yFirstDisplay 378  
yFirstDualPower 456  
yFirstFiles 481  
yFirstGenericSensor 509  
yFirstGyro 555  
yFirstHubPort 605  
yFirstHumidity 631  
yFirstLed 669  
yFirstLightSensor 697  
yFirstMagnetometer 737  
yFirstModule 785  
yFirstNetwork 822  
yFirstOsControl 877  
yFirstPower 901  
yFirstPressure 944  
yFirstPwmOutput 983  
yFirstPwmPowerSource 1020  
yFirstQt 1044  
yFirstRealTimeClock 1082  
yFirstRefFrame 1110  
yFirstRelay 1146  
yFirstSensor 1182  
yFirstServo 1221  
yFirstTemperature 1256  
yFirstTilt 1297  
yFirstVoc 1336  
yFirstVoltage 1375  
yFirstVSource 1413  
yFirstWakeUpMonitor 1446  
yFirstWakeUpSchedule 1481  
yFirstWatchdog 1518  
yFirstWireless 1562  
yFreeAPI 20  
YGenericSensor 508-550  
yGetAPIVersion 21

yGetTickCount 22  
YGyro 554-601  
yHandleEvents 23  
YHubPort 604-626  
YHumidity 630-665  
yInitAPI 24  
YLed 668-692  
YLightSensor 696-732  
YMagnetometer 736-774  
YMeasure 776-780  
YModule 784-816  
YNetwork 821-873  
Yocto-Demo 3  
Yocto-hub 603  
YOsControl 876-896  
YPower 900-939  
yPreregisterHub 25  
YPressure 943-978  
YPwmOutput 982-1016  
YPwmPowerSource 1019-1039  
YQt 1043-1078  
YRealTimeClock 1081-1105  
YRefFrame 1109-1141  
yRegisterDeviceArrivalCallback 26  
yRegisterDeviceRemovalCallback 27  
yRegisterHub 28  
yRegisterHubDiscoveryCallback 29  
yRegisterLogFunction 30  
YRelay 1145-1177  
YSensor 1181-1216  
YServo 1220-1251  
ySleep 31  
YTemperature 1255-1292  
YTilt 1296-1331  
yTriggerHubDiscovery 32  
yUnregisterHub 33  
yUpdateDeviceList 34  
YVoc 1335-1370  
YVoltage 1374-1409  
YVSource 1412-1441  
YWakeUpMonitor 1445-1476  
YWakeUpSchedule 1480-1513  
YWatchdog 1517-1558  
YWireless 1561-1587