# Table of contents

1. Introduction .............................................................................................................................. 1

2. Using Yocto-Demo with C# .................................................................................................... 3
   2.1. Installation .......................................................................................................................... 3
   2.2. Using the Yoctopuce API in a Visual C# project .............................................................. 3
   2.3. Control of the Led function ............................................................................................... 4
   2.4. Control of the module part ............................................................................................... 5
   2.5. Error handling ................................................................................................................... 8
   Blueprint ..................................................................................................................................... 12

3. Reference ................................................................................................................................... 12
   3.1. General functions ............................................................................................................. 13
   3.2. Accelerometer function interface .................................................................................. 33
   3.3. Altitude function interface ............................................................................................ 75
   3.4. AnButton function interface ......................................................................................... 117
   3.5. CarbonDioxide function interface .............................................................................. 155
   3.6. ColorLed function interface ......................................................................................... 194
   3.7. Compass function interface ........................................................................................ 223
   3.8. Current function interface ............................................................................................. 263
   3.9. DataLogger function interface ....................................................................................... 302
   3.10. Formatted data sequence .............................................................................................. 336
   3.11. Recorded data sequence ............................................................................................... 338
   3.12. Unformatted data sequence ......................................................................................... 350
   3.13. Digital IO function interface ....................................................................................... 365
   3.14. Display function interface ............................................................................................ 409
   3.15. DisplayLayer object interface .................................................................................... 456
   3.16. External power supply control interface ................................................................... 488
   3.17. Files function interface ................................................................................................. 513
   3.18. GenericSensor function interface ............................................................................... 540
   3.19. Gyroscope function interface ....................................................................................... 589
   3.20. Yocto-hub port interface .............................................................................................. 640
   3.21. Humidity function interface ......................................................................................... 665
   3.22. Led function interface .................................................................................................. 704
   3.23. LightSensor function interface ..................................................................................... 731
   3.24. Magnetometer function interface ............................................................................... 773
3.25. Measured value .............................................................................................................. 815
3.26. Module control interface .......................................................................................... 821
3.27. Motor function interface ......................................................................................... 865
3.28. Network function interface ...................................................................................... 906
3.29. OS control ............................................................................................................. 963
3.30. Power function interface ........................................................................................ 986
3.31. Pressure function interface .................................................................................... 1029
3.32. PwmInput function interface .................................................................................. 1068
3.33. Pwm function interface .......................................................................................... 1116
3.34. PwmPowerSource function interface .................................................................... 1154
3.35. Quaternion interface ..................................................................................................... 1177
3.36. Real Time Clock function interface .......................................................................... 1216
3.37. Reference frame configuration .................................................................................. 1243
3.38. Relay function interface ........................................................................................ 1279
3.39. Sensor function interface ........................................................................................ 1315
3.40. SerialPort function interface .................................................................................... 1354
3.41. Servo function interface .......................................................................................... 1411
3.42. Temperature function interface ................................................................................ 1446
3.43. Tilt function interface .................................................................................................. 1487
3.44. Voc function interface .............................................................................................. 1526
3.45. Voltage function interface ......................................................................................... 1565
3.46. Voltage source function interface ............................................................................ 1604
3.47. WakeUpMonitor function interface .......................................................................... 1636
3.48. WakeUpSchedule function interface ...................................................................... 1671
3.49. Watchdog function interface .................................................................................... 1708
3.50. Wireless function interface ....................................................................................... 1753

Index ......................................................................................................................................... 1783
1. Introduction

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

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

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

C# (pronounced C-Sharp) is an object-oriented programming language promoted by Microsoft, it is somewhat similar to Java. Like Visual-Basic and Delphi, it allows you to create Windows applications quite easily. All the examples and the project models are tested with Microsoft C# 2010 Express, freely available on the Microsoft web site\(^1\).

2.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site\(^2\). There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the Sources directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual C# 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

2.2. Using the Yoctopuce API in a Visual C# project

The Visual C#.NET Yoctopuce library is composed of a DLL and of source files in Visual C#. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules\(^3\). The source files in Visual C# manage the high level part of the API. Therefore, your need both this DLL and the .cs files of the sources directory to create a project managing Yoctopuce modules.

Configuring a Visual C# project

The following indications are provided for Visual Studio Express 2010, but the process is similar for other versions. Start by creating your project. Then, on the Solution Explorer panel, right click on your project, and select "Add" and then "Add an existing item".

A file selection window opens. Select the yocto_api.cs file and the files corresponding to the functions of the Yoctopuce modules that your project is going to manage. If in doubt, select all the files.

You then have the choice between simply adding these files to your project, or to add them as links (the Add button is in fact a scroll-down menu). In the first case, Visual Studio copies the selected files into your project. In the second case, Visual Studio simply keeps a link on the original files. We recommend you to use links, which makes updates of the library much easier.

---

\(^2\) www.yoctopuce.com/EN/libraries.php
\(^3\) The sources of this DLL are available in the C++ API
Then add in the same manner the yapi.dll DLL, located in the Sources/dll directory. Then, from the Solution Explorer window, right click on the DLL, select Properties and in the Properties panel, set the Copy to output folder to always. You are now ready to use your Yoctopuce modules from Visual Studio.

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

2.3. Control of the Led function

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

```csharp
[...] string errmsg="";
YLed led;

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb", errmsg);
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's look at these lines in more details.

YAPI.RegisterHub

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

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.

```csharp
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

---

4 Remember to change the filter of the selection window, otherwise the DLL will not show.
programming interface, you will find more methods to precisely control the luminosity and make the led blink automatically.

**A real example**

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory *Examples/Doc-GettingStarted-Yocto-Demo* of the Yoctopuce library.

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

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void usage()
        {
            string execname = System.AppDomain.CurrentDomain.FriendlyName;
            Console.WriteLine(execname + " <serial_number> [ on | off ] ");
            Console.WriteLine(execname + " <logical_name> [ on | off ] ");
            Console.WriteLine(execname + " any [ on | off ] ");
            System.Threading.Thread.Sleep(2500);
            Environment.Exit(0);
        }

        static void Main(string[] args)
        {
            string errmsg = ";
            string target;
            YLed led;
            string on_off;

            if (args.Length < 2) usage();
            target = args[0].ToUpper();
            on_off = args[1].ToUpper();

            if (YAPI.RegisterHub("usb", ref errmsg) != YAPI.SUCCESS)
            {
                Console.WriteLine("RegisterHub error: " + errmsg);
                Environment.Exit(0);
            }

            if (target == "ANY")
            {
                led = YLed.FirstLed();
                if (led == null)
                {
                    Console.WriteLine("No module connected (check USB cable) ");
                    Environment.Exit(0);
                }
            }
            else led = YLed.FindLed(target + ".led");

            if (led.isOnline())
            {
                if (on_off == "ON") led.set_power(YLed.POWER_ON); else led.set_power(YLed.POWER_OFF);
            }
            else Console.WriteLine("Module not connected (check identification and USB cable) ");
        }
    }
}
```

**2.4. Control of the module part**

Each module can be controlled in a similar manner, you can find below a simple sample program displaying the main parameters of the module and enabling you to activate the localization beacon.
```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ...
{
    class Program
    {
        static void usage()
        {
            string execname = System.AppDomain.CurrentDomain.FriendlyName;
            Console.WriteLine("Usage: ");
            Console.WriteLine(execname + " <serial or logical name> [ON/OFF] ");
            System.Threading.Thread.Sleep(2500);
            Environment.Exit(0);
        }

        static void Main(string[] args)
        {
            YModule m;
            string errmsg = ";
            if (YAPI.RegisterHub("usb", ref errmsg) != YAPI.SUCCESS)
            {
                Console.WriteLine("RegisterHub error: " + errmsg);
                Environment.Exit(0);
            }

            if (args.Length < 1) usage();

            m = YModule.FindModule(args[0]); // use serial or logical name
            if (m.isOnline())
            {
                if (args.Length >= 2)
                {
                    if (args[1].ToUpper() == "ON") { m.set_beacon(YModule.BEACON_ON); }
                    if (args[1].ToUpper() == "OFF") { m.set_beacon(YModule.BEACON_OFF); }
                }

                Console.WriteLine("serial: " + m.get_serialNumber());
                Console.WriteLine("logical name: " + m.get_logicalName());
                Console.WriteLine("luminosity: " + m.get_luminosity().ToString());
                Console.WriteLine("beacon: ");
                if (m.get_beacon() == YModule.BEACON_ON)
                    Console.WriteLine("ON");
                else
                    Console.WriteLine("OFF");
                Console.WriteLine("upTime: " + (m.get_upTime() / 1000).ToString() + " sec");
                Console.WriteLine("USB current: " + m.get_usbCurrent().ToString() + " mA");
                Console.WriteLine("Logs:\r\n" + m.get_lastLogs());
            }
            else
                Console.WriteLine(args[0] + " not connected (check identification and USB cable) ");
        }
    }
}
```

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_xxx()`, 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.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void usage()
        {
            string execname = System.AppDomain.CurrentDomain.FriendlyName;
            Console.WriteLine("usage: demo <serial or logical name> <new logical name>");
            Environment.Exit(0);
        }

        static void Main(string[] args)
        {
            YModule m;
            string errmsg = "";
            string newname;

            if (args.Length != 2) usage();

            if (!YAPI.RegisterHub("usb", ref errmsg) != YAPI.SUCCESS)
            {
                Console.WriteLine("RegisterHub error: "+errmsg);
                Environment.Exit(0);
            }

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

            if (m.isOnline())
            {
                newname = args[1];
                if (!YAPI.CheckLogicalName(newname))
                {
                    Console.WriteLine("Invalid name (" + newname + ")");
                    Environment.Exit(0);
                }

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

                Console.WriteLine("Module: serial= " + m.get_serialNumber());
                Console.WriteLine(" / name= " + m.get_logicalName());
            }
            else
            {
                Console.WriteLine("not connected (check identification and USB cable)");
            }
        }
    }
}
```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees 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.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;

```
2.5. Error handling

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

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

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

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

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

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

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

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

```nodejs
var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
require_once('yocto_api.php');
#include "yocto_api.h"
import "yocto_api.h"
uses yocto_api;
yocto_api.vb
yocto_api.cs
import com.yoctopuce.YoctoAPI.YModule;
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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yRegisterLogFunction(logfun)</code></td>
<td>Registers a log callback function.</td>
</tr>
<tr>
<td><code>ySelectArchitecture(arch)</code></td>
<td>Select the architecture or the library to be loaded to access to USB.</td>
</tr>
<tr>
<td><code>ySetDelegate(object)</code></td>
<td>(Objective-C only) Register an object that must follow the protocol <code>YDeviceHotPlug</code>.</td>
</tr>
<tr>
<td><code>ySetTimeout(callback, ms_timeout, arguments)</code></td>
<td>Invoke the specified callback function after a given timeout.</td>
</tr>
<tr>
<td><code>ySleep(ms_duration, errmsg)</code></td>
<td>Pauses the execution flow for a specified duration.</td>
</tr>
<tr>
<td><code>yTriggerHubDiscovery(errmsg)</code></td>
<td>Force a hub discovery, if a callback as been registered with <code>yRegisterDeviceRemovalCallback</code> it will be called for each net work hub that will respond to the discovery.</td>
</tr>
<tr>
<td><code>yUnregisterHub(url)</code></td>
<td>Setup the Yoctopuce library to no more use modules connected on a previously registered machine with RegisterHub.</td>
</tr>
<tr>
<td><code>yUpdateDeviceList(errmsg)</code></td>
<td>Triggers a (re)detection of connected Yoctopuce modules.</td>
</tr>
<tr>
<td><code>yUpdateDeviceList_async(callback, context)</code></td>
<td>Triggers a (re)detection of connected Yoctopuce modules.</td>
</tr>
</tbody>
</table>
YAPI.CheckLogicalName()

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

```c
bool 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.
3. Reference

YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

```c
void DisableExceptions()
```

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

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

```c
void EnableExceptions( )
```

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

Frees dynamically allocated memory blocks used by the Yoctopuce library.

```c
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()
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()  
yGetTickCount()  
YAPI.GetTickCount()

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

ulong 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()**

Maintains the device-to-library communication channel.

**YRETCODE HandleEvents( ref string errmsg)***

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

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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errmsg</code></td>
<td>a string passed by reference to receive any error message.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS when the call succeeds.</td>
<td></td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
YAPI.InitAPI()
yInitAPI()YAPI.InitAPI()

Initializes the Yoctopuce programming library explicitly.

```c
int InitAPI( int mode, ref string errmsg)
```

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

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

**Parameters :**
- `mode` an integer corresponding to the type of automatic device detection to use. Possible values are `Y_DETECT_NONE`, `Y_DETECT_USB`, `Y_DETECT_NET`, and `Y_DETECT_ALL`.
- `errmsg` a string passed by reference to receive any error message.

**Returns :**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
YAPI.PreregisterHub()

Fault-tolerant alternative to RegisterHub().

```c
int PreregisterHub(string url, ref string errmsg)
```

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

**Parameters:**
- `url`  a string containing either "usb", "callback" or the root URL of the hub to monitor
- `errmsg` a string passed by reference to receive any error message.

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
YAPI registradoDeviceArrivalCallback()    
yRegisterDeviceArrivalCallback()    
YAPI registradoDeviceArrivalCallback()

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

```c
void RegisterDeviceArrivalCallback( yDeviceUpdateFunc arrivalCallback)
```

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

**Parameters:**

- **arrivalCallback** a procedure taking a YModule parameter, or null
YAPI.RegisterDeviceRemovalCallback()
yRegisterDeviceRemovalCallback()
YAPI.RegisterDeviceRemovalCallback()

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

```c
void RegisterDeviceRemovalCallback( yDeviceUpdateFunc removalCallback)
```

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

**Parameters:**

- `removalCallback` a procedure taking a `YModule` parameter, or null
YAPI.RegisterHub()
yRegisterHub()YAPI.RegisterHub()

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

```c
int RegisterHub( string url, ref string errmsg)
```

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

**usb**: When the **usb** keyword is used, the API will work with devices connected directly to the USB bus. Some programming languages such 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**: this keyword make the API run in "HTTP Callback" mode. This a special mode allowing to take control of Yoctopuce devices through a NAT filter when using a VirtualHub or a networked YoctoHub. You only need to configure your hub to call your server script on a regular basis. This mode is currently available for PHP and Node.JS only.

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

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

http://username:password@address:port

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

**Parameters**:
- **url**: a string containing either "usb","callback" or the root URL of the hub to monitor
- **errmsg**: a string passed by reference to receive any error message.

**Returns**:
- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
YAPI.RegisterHubDiscoveryCallback()
yRegisterHubDiscoveryCallback()
YAPI.RegisterHubDiscoveryCallback()

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

```c
void RegisterHubDiscoveryCallback( YHubDiscoveryCallback 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()  

Registers a log callback function.

```c
void RegisterLogFunction( yLogFunc 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()
ySleep() YAPI.Sleep()

Pauses the execution flow for a specified duration.

```c
int Sleep( int ms_duration, ref string errmsg)
```

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

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

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

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
YAPI.TriggerHubDiscovery()

**YAPI**

**yTriggerHubDiscovery()**

YAPI.TriggerHubDiscovery()

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

```csharp
int TriggerHubDiscovery( ref string errmsg)
```

**Parameters :**
- `errmsg` a string passed by reference to receive any error message.

**Returns :**
- `YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error
code.
YAPI.UnregisterHub()
yUnregisterHub() YAPI.UnregisterHub()

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

```c
void UnregisterHub( string url)
```

**Parameters:**
- `url` a string containing either "usb" or the
3. Reference

YAPI.UpdateDeviceList()  

yUpdateDeviceList()  

YAPI.UpdateDeviceList()

<table>
<thead>
<tr>
<th>YAPI</th>
<th>YAPI</th>
<th>YAPI</th>
</tr>
</thead>
</table>

Triggers a (re)detection of connected Yoctopuce modules.

```
YRETCODE UpdateDeviceList( ref string errmsg)
```

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

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

**Parameters:**
- `errmsg` a string passed by reference to receive any error message.

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

The 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YAccelerometer = yoctolib.YAccelerometer;
require_once('yocto_accelerometer.php');
```

```cpp
#include "yocto_accelerometer.h"
```

```m
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, in g, as a floating point number.

#### `accelerometer → get_currentValue()`
Returns the current value of the acceleration, in g, as a floating point number.

#### `accelerometer → get_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`. 
3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_highestValue()</code></td>
<td>Returns the maximal value observed for the acceleration since the device was started.</td>
</tr>
<tr>
<td><code>get_lowestValue()</code></td>
<td>Returns the minimal value observed for the acceleration since the device was started.</td>
</tr>
<tr>
<td><code>get_logicalName()</code></td>
<td>Returns the logical name of the accelerometer.</td>
</tr>
<tr>
<td><code>get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>get_recordedData(startTime, endTime)</code></td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><code>get_reportFrequency()</code></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><code>get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>get_unit()</code></td>
<td>Returns the measuring unit for the acceleration.</td>
</tr>
<tr>
<td><code>get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>get_xValue()</code></td>
<td>Returns the X component of the acceleration, as a floating point number.</td>
</tr>
<tr>
<td><code>get_yValue()</code></td>
<td>Returns the Y component of the acceleration, as a floating point number.</td>
</tr>
<tr>
<td><code>get_zValue()</code></td>
<td>Returns the Z component of the acceleration, as a floating point number.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the accelerometer is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the accelerometer is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the accelerometer cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the accelerometer cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextAccelerometer()</code></td>
<td>Continues the enumeration of accelerometers started using <code>yFirstAccelerometer()</code>.</td>
</tr>
<tr>
<td><code>registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the accelerometer.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>accelerometer-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
3. Reference

YAccelerometer.FindAccelerometer()

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()**

Starts the enumeration of accelerometers currently accessible.

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\rightarrow{\text{calibrateFromPoints}}()

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

```
int calibrateFromPoints(List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
accelerometer.describe()

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

```
string describe()
```

More precisely, \text{TYPE} is the type of the function, \text{NAME} it the name used for the first access to the function, \text{SERIAL} is the serial number of the module if the module is connected or "unresolved", and \text{FUNCTIONID} is the hardware identifier of the function if the module is connected. For example, this method returns \text{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1} if the module is already connected or \text{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: \text{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1})
```
3. Reference

`accelerometer -> get_advertisedValue()`

`accelerometer -> advertisedValue()`

`accelerometer.get_advertisedValue()`

YAccelerometer

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, in g, as a floating point number.

**Returns:**

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>accelerometer.get_currentValue()</code></td>
<td><code>double</code></td>
<td>Returns the current value of the acceleration, in g, as a floating point number.</td>
</tr>
</tbody>
</table>
| `accelerometer.currentValue()`      |             | Returns:
<p>|                                   |             | a floating point number corresponding to the current value of the acceleration, in g, as a floating point number |
| On failure, throws an exception or returns <code>Y_CURRENTVALUE_INVALID</code>. |             |             |</p>
<table>
<thead>
<tr>
<th>accelerometer → get_errorMessage()</th>
<th>YAccelerometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>accelerometer → errorMessage()</td>
<td></td>
</tr>
<tr>
<td>accelerometer.get_errorMessage()</td>
<td></td>
</tr>
</tbody>
</table>

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

```cpp
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.

YRETCode 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() \hspace{2cm} YAccelerometer
accelerometer→friendlyName()
accelerometer.get_friendlyName()

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

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

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

```cpp
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`. 
3. Reference

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

YAccelerometer

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()
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()
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()  
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()
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
3. Reference

**accelerometer→get_recordedData()**
**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 measure, 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 measure, 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.
3. Reference

YAccelerometer

accelerometer → get_reportFrequency()
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 YREPORTFREQUENCY_INVALID.
accelerometer\texttt{\rightarrow get\_resolution()}
accelerometer\texttt{\rightarrow resolution()}
accelerometer\texttt{.get\_resolution()}

Returns the resolution of the measured values.

\texttt{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.

\textbf{Returns :}

- a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns \texttt{Y\_RESOLUTION\_INVALID}.
accelerometer→get_unit()
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.
3. Reference

```
accelerometer→get_userData()
accelerometer→userData()
accelerometer.get_userData()
```

YAccelerometer

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()
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.
3. Reference

YAccelerometer

accelerometer→get_yValue()
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.
3. Reference

**YAccelerometer**

`accelerometer→get_zValue()`

`accelerometer→zValue()`

`accelerometer.get_zValue()`

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the Z component of the acceleration, as a floating point number</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_ZVALUE_INVALID.
accelerometer → isOnline() → accelerometer.isOnline()  

**YAccelerometer**

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

```cpp
bool 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()

Preloads the accelerometer cache with a specified validity duration.

**YRETCODE load(int 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.
3. Reference

YAccelerometer
accelerometer.loadCalibrationPoints()

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
<table>
<thead>
<tr>
<th>YAccelerometer</th>
<th>accelerometer.nextAccelerometer()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Continues the enumeration of accelerometers started using yFirstAccelerometer().</td>
</tr>
</tbody>
</table>

**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.
### 3. Reference

**accelerometer.registerTimedReportCallback()**

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

```c
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

YAccelerometer

accelerometer.registerValueCallback()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
Changes the recorded maximal value observed.

```c
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()
accelerometer->setLogFrequency()
accelerometer.set_logFrequency()

Changes the datalogger recording frequency for this function.

```c
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.
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.
Changes the recorded minimal value observed.

```c
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()**  
**accelerometer → setReportFrequency()**  
**accelerometer.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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()
accelerometer→setUserData()
accelerometer.set_userData()

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

```cpp
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. Altitude 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:

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

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindAltitude()</code></td>
<td>Retrieves an altimeter for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstAltitude()</code></td>
<td>Starts the enumeration of altimeters currently accessible.</td>
</tr>
</tbody>
</table>

### YAltitude methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calibrateFromPoints()</code></td>
<td>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.</td>
</tr>
<tr>
<td><code>describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the altimeter in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>get_advertisedValue()</code></td>
<td>Returns the current value of the altimeter (no more than 6 characters).</td>
</tr>
<tr>
<td><code>get_currentRawValue()</code></td>
<td>Returns the uncalibrated, unrounded raw value returned by the sensor, in meters, as a floating point number.</td>
</tr>
<tr>
<td><code>get_currentValue()</code></td>
<td>Returns the current value of the altitude, in meters, as a floating point number.</td>
</tr>
<tr>
<td><code>get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the altimeter.</td>
</tr>
<tr>
<td><code>get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the altimeter.</td>
</tr>
<tr>
<td><code>get_friendlyName()</code></td>
<td>Returns a global identifier of the altimeter in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><code>get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><code>get_functionId()</code></td>
<td>Returns the hardware identifier of the altimeter, without reference to the module.</td>
</tr>
<tr>
<td><code>get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the altimeter in the form SERIAL.FUNCTIONID.</td>
</tr>
</tbody>
</table>
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>altitude.get_highestValue()</td>
<td>Returns the maximal value observed for the altitude since the device was started.</td>
</tr>
<tr>
<td>altitude.get_logFrequency()</td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td>altitude.get_logicalName()</td>
<td>Returns the logical name of the altimeter.</td>
</tr>
<tr>
<td>altitude.get_lowestValue()</td>
<td>Returns the minimal value observed for the altitude since the device was started.</td>
</tr>
<tr>
<td>altitude.get_module()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>altitude.get_module_async()</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>altitude.get_qnh()</td>
<td>Returns the barometric pressure adjusted to sea level used to compute the altitude (QNH).</td>
</tr>
<tr>
<td>altitude.get_recordedData()</td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td>altitude.get_reportFrequency()</td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td>altitude.get_resolution()</td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td>altitude.get_unit()</td>
<td>Returns the measuring unit for the altitude.</td>
</tr>
<tr>
<td>altitude.get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>altitude.isOnline()</td>
<td>Checks if the altimeter is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>altitude.isOnline_async()</td>
<td>Checks if the altimeter is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>altitude.load()</td>
<td>Preloads the altimeter cache with a specified validity duration.</td>
</tr>
<tr>
<td>altitude.loadCalibrationPoints()</td>
<td>Retrieves error correction data points previously entered using the method calibrateFromPoints.</td>
</tr>
<tr>
<td>altitude.load_async()</td>
<td>Preloads the altimeter cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>altitude.nextAltitude()</td>
<td>Continues the enumeration of altimeters started using yFirstAltitude().</td>
</tr>
<tr>
<td>altitude.registerTimedReportCallback()</td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td>altitude.registerValueCallback()</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>altitude.set_currentValue(newval)</td>
<td>Changes the current estimated altitude.</td>
</tr>
<tr>
<td>altitude.set_highestValue(newval)</td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>altitude→set_logFrequency(newval)</td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td>altitude→set_logicalName(newval)</td>
<td>Changes the logical name of the altimeter.</td>
</tr>
<tr>
<td>altitude→set_lowestValue(newval)</td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td>altitude→set_qnh(newval)</td>
<td>Changes the barometric pressure adjusted to sea level used to compute the altitude (QNH).</td>
</tr>
<tr>
<td>altitude→set_reportFrequency(newval)</td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td>altitude→set_resolution(newval)</td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td>altitude→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>altitude→wait_async(callback, context)</td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YAltitude.FindAltitude

YAltitude.FindAltitude(string func)

Retrieves an altimeter for a given identifier.

The identifier can be specified using several formats:

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

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

**Parameters:**

- **func** a string that uniquely characterizes the altimeter

**Returns:**

- a YAltitude object allowing you to drive the altimeter.
YAltitude.FirstAltitude()

YAltitude.FirstAltitude()

YAltitude.FirstAltitude()

YAltitude.FirstAltitude()

YAltitude.FirstAltitude()

Starts the enumeration of altimeters currently accessible.

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

Returns:

- a pointer to a YAltitude object, corresponding to the first altimeter currently online, or a null pointer if there are none.
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```c
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
**altitude.describe()**

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

```csharp
string describe()
```

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

**Returns:**

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

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

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

string get_advertisedValue()

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string corresponding to the current value of the altimeter (no more than 6 characters).</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
Returns the uncalibrated, unrounded raw value returned by the sensor, in meters, as a floating point number.

**Returns**:

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

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.
### 3. Reference

**YAltitude**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>altitude.get_currentValue()</code></td>
<td>Returns the current value of the altitude, in meters, as a floating point number.</td>
</tr>
</tbody>
</table>

**Returns:**

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

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

```
altitude.get_errorMessage()
altitude.errorMessage() altitude.get_errorMessage()
```

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

```
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 altimeter object
YAltitude

altitude\rightarrow get\_errorType()

altitude\rightarrow errorType() \rightarrow altitude.get\_errorType()

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

\textbf{YRETCODE get\_errorType()}( )

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

\textbf{Returns :}

a number corresponding to the code of the latest error that occurred while using the altimeter object
altitude\rightarrow get\_friendlyName()

\textbf{YAltitude} \quad\textbf{altitude} \rightarrow \textbf{friendlyName()} \quad \textbf{altitude.get\_friendlyName()}

Returns a global identifier of the altimeter in the format \texttt{MODULE\_NAME.FUNCTION\_NAME}.

\textbf{string get\_friendlyName()}( )

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

Returns :

- a string that uniquely identifies the altimeter using logical names (ex: \texttt{MyCustomName.relay1})

On failure, throws an exception or returns \texttt{Y\_FRIENDLYNAME\_INVALID}. 

YAltitude

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

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

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

```
string get_functionId()
```

For example `relay1`

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

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

3. Reference

Returns the unique hardware identifier of the altimeter 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 altimeter (for example RELAYLO1-123456.relay1).

Returns :

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
<table>
<thead>
<tr>
<th>altitude→get_highestValue()</th>
<th>Returns the maximal value observed for the altitude since the device was started.</th>
</tr>
</thead>
<tbody>
<tr>
<td>altitude→highestValue()</td>
<td>Returns :</td>
</tr>
<tr>
<td>altitude.get_highestValue()</td>
<td>a floating point number corresponding to the maximal value observed for the altitude since the device was started</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.</td>
</tr>
</tbody>
</table>
YAltitude

altitude → get_logFrequency()
alogue → logFrequency() altitude.get_logFrequency()

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

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

Returns the logical name of the altimeter.

String get_logicalName( )

Returns:

a string corresponding to the logical name of the altimeter.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
Returns the minimal value observed for the altitude since the device was started.

Returns:

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

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

\texttt{altitude.get\_module()}
\texttt{altitude.get\_module()}

Gets the \texttt{YModule} object for the device on which the function is located.

\texttt{YModule get\_module()}

If the function cannot be located on any module, the returned instance of \texttt{YModule} is not shown as online.

**Returns:**

an instance of \texttt{YModule}
Returns the barometric pressure adjusted to sea level used to compute the altitude (QNH).

double get_qnh()

Returns:

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

On failure, throws an exception or returns Y_QNH_INVALID.
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```python
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 measure, 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 measure, 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.
<table>
<thead>
<tr>
<th>YAltitude</th>
<th>altitude.get_reportFrequency()</th>
</tr>
</thead>
<tbody>
<tr>
<td>altitude</td>
<td>reportFrequency()</td>
</tr>
<tr>
<td>altitude</td>
<td>get_reportFrequency()</td>
</tr>
</tbody>
</table>

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

```cpp
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.
}
```
altitude\rightarrow \text{get\_resolution()}
altitude\rightarrow \text{resolution()}
altitude.get\_resolution()

Returns the resolution of the measured values.

double \text{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 $Y\_\text{RESOLUTION\_INVALID}$. 
YAltitude

altitude unit() altitude.get_unit()

Returns the measuring unit for the altitude.

string get_unit()

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

On failure, throws an exception or returns `Y_UNIT_INVALID`. 
altitude.get_userData() YAltitude altitude.getUserData()altitude.getUserData() altitude.getUserData() altitude.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.
altitude->isOnline()\hspace{1cm} YAltitude

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

```c
bool isOnline()
```

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

**Returns:**
- `true` if the altimeter can be reached, and `false` otherwise
Preloads the altimeter cache with a specified validity duration.

**YRETCODE load( int msValidity)**

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

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

**Returns :**
- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
altitude\rightarrow\text{loadCalibrationPoints()}

\text{altitude.loadCalibrationPoints()}

Retrieves error correction data points previously entered using the method \text{calibrateFromPoints}.

\begin{verbatim}
int loadCalibrationPoints( List<double> rawValues, List<double> refValues)
\end{verbatim}

\textbf{Parameters :}
- \textbf{rawValues} array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- \textbf{refValues} array of floating point numbers, that will be filled by the function with the desired values for the correction points.

\textbf{Returns :}
- \textbf{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the enumeration of altimeters started using `yFirstAltitude()`.

**YAltitude nextAltitude()**

**Returns:**

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

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

```c
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.
altitude\rightarrow\text{registerValueCallback()}

\text{altitude.registerValueCallback()}

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

\text{int registerValueCallback( ValueCallback callback)}

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

\textbf{Parameters :}

\begin{itemize}
  \item \text{callback} the callback 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.
\end{itemize}
### Changes the current estimated altitude.

```c
int set_currentValue( double newval)
```

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

**Parameters:**
- `newval` a floating point number corresponding to the current estimated altitude

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

On failure, throws an exception or returns a negative error code.
Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
int set_logFrequency( string newval)
```

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

**Parameters:**
- `newval` a string corresponding to the datalogger recording frequency for this function

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

On failure, throws an exception or returns a negative error code.
altitude.set_logicalName()

Changes the logical name of the altimeter.

```c
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 altimeter.

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

**YAltitude**

```c
altitude->set_lowestValue()
```

Changes the recorded minimal value observed.

```c
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.
YAltitude

```
altitude->set_qnh()
altitude->setQnh() altitude.set_qnh()
```

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

```
int set_qnh( double newval)
```

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

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

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

On failure, throws an exception or returns a negative error code.
altitude.set_reportFrequency()

Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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.
3. Reference

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

```c
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. 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:

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

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindAnButton</strong>&lt;sup&gt;(func)&lt;/sup&gt;</td>
</tr>
<tr>
<td><strong>yFirstAnButton</strong>&lt;sup&gt;()&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>YAnButton methods</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>anbutton→describe()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_advertisedValue()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_analogCalibration()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_calibratedValue()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_calibrationMax()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_calibrationMin()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_errorMessage()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_errorType()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_friendlyName()</strong></td>
</tr>
<tr>
<td><strong>anbutton→get_functionDescriptor()</strong></td>
</tr>
</tbody>
</table>
### 3. Reference

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

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

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

- **anbutton→get_logicalName()**
  Returns the logical name of the analog input.

- **anbutton→get_module()**
  Gets the YModule object for the device on which the function is located.

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

- **anbutton→get_pulseCounter()**
  Returns the pulse counter value.

- **anbutton→get_pulseTimer()**
  Returns the timer of the pulses counter (ms).

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

- **anbutton→get_sensitivity()**
  Returns the sensibility for the input (between 1 and 1000) for triggering user callbacks.

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

- **anbutton→isOnline()**
  Checks if the analog input is currently reachable, without raising any error.

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

- **anbutton→load(msValidity)**
  Preloads the analog input cache with a specified validity duration.

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

- **anbutton→nextAnButton()**
  Continues the enumeration of analog inputs started using `yFirstAnButton()`.

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

- **anbutton→resetCounter()**
  Returns the pulse counter value as well as his timer.

- **anbutton→set_analogCalibration(newval)**
  Starts or stops the calibration process.

- **anbutton→set_calibrationMax(newval)**
Changes the maximal calibration value for the input (between 0 and 4095, included), without actually starting
the automated calibration.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anbutton-&gt;set_calibrationMin(newval)</code></td>
<td>Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.</td>
</tr>
<tr>
<td><code>anbutton-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the analog input.</td>
</tr>
<tr>
<td><code>anbutton-&gt;set_sensitivity(newval)</code></td>
<td>Changes the sensibility for the input (between 1 and 1000) for triggering user callbacks.</td>
</tr>
<tr>
<td><code>anbutton-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>anbutton-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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()

yFirstAnButton() YAnButton.FirstAnButton()

Starts the enumeration of analog inputs currently accessible.

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

Returns:
a pointer to a YAnButton object, corresponding to the first analog input currently online, or a null pointer if there are none.
anbutton.describe() Returns a short text that describes unambiguously the instance of the analog input in the form \texttt{TYPE(NAME)=SERIAL.FUNCTIONID}.

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

Returns :

\begin{itemize}
  \item a string that describes the analog input (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
\end{itemize}
anbutton→get_advertisedValue()
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.
### 3. Reference

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

Tells if a calibration process is currently ongoing.

```c
int get_analogCalibration()
```

**Returns:**

- either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

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

YAnButton

anbutton→get_calibrationMax()
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.
### YAnButton

- **get_calibrationMin()**
- **calibrationMin()**
- **anbutton.get_calibrationMin()**

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

```c
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.
YAnButton

anbutton→get_errorMessage()
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()
anbutton → errorType() anbutton.get_errorType()

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

YRETCODE 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
YAnButton

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

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

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

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
anbutton→get_functionDescriptor()

YAnButton

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

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

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

```java
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`. 
| 3. Reference |

**YAnButton**

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

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

```c
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`. 
YAnButton

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

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 transitioned from open to closed)

On failure, throws an exception or returns Y_LASTTIMEPRESSED_INVALID.
YAnButton

anbutton \rightarrow \text{get\_lastTimeReleased()}
anbutton \rightarrow \text{lastTimeReleased()}
anbutton.get\_lastTimeReleased()

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

\text{long get\_lastTimeReleased( )}

\textbf{Returns :}
  \text{an integer corresponding to the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitioned from closed to open)}

\text{On failure, throws an exception or returns Y\_LASTTIMERELEASED\_INVALID.}
YAnButton

anbutton->get_logicalName()
anbutton->logicalName()anbutton.get_logicalName()

Returns the logical name of the analog input.

```c
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
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.
YAnButton

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

Returns the timer of the pulses counter (ms)

long get_pulseTimer( )

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

On failure, throws an exception or returns Y_PULSETIMER_INVALID.
YAnButton

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

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

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.
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.
YAnButton

anbutton→isOnline()anbutton.isOnline()

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

```cpp
bool 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()

Preloads the analog input cache with a specified validity duration.

YReturnType load( int 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.
3. Reference

**anbutton.nextAnButton()**

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

**Returns:**

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

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

```c
int registerValueCallback(ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
YAnButton

```
anbutton.resetCounter()
```

Returns the pulse counter value as well as his timer

```c
int resetCounter()
```

Returns:

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
anbutton→set_analogCalibration()
anbutton→setAnalogCalibration()
anbutton.set_analogCalibration()

Starts or stops the calibration process.

```c
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.
3. Reference

`YAnButton anbutton;`  

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

```c
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()

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

```c
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.
Changes the logical name of the analog input.

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

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

```c
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. 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:

```javascript
<script type='text/javascript' src='yocto_carbondioxide.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YCarbonDioxide = yoctolib.YCarbonDioxide;
```
```php
require_once('yocto_carbondioxide.php');
```
```cpp
#include "yocto_carbondioxide.h"
```
```m
#include "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 *
```

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindCarbonDioxide(func)</strong></td>
</tr>
<tr>
<td>Retrieves a CO2 sensor for a given identifier.</td>
</tr>
</tbody>
</table>

| **yFirstCarbonDioxide()** |
| Starts the enumeration of CO2 sensors currently accessible. |

<table>
<thead>
<tr>
<th><strong>YCarbonDioxide methods</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>carbondioxide→calibrateFromPoints(rawValues, refValues)</strong></td>
</tr>
<tr>
<td>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.</td>
</tr>
</tbody>
</table>

| **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, in ppm (vol), as a floating point number. |

| **carbondioxide→get_currentValue()** |
| Returns the current value of the CO2 concentration, in ppm (vol), as a floating point number. |

| **carbondioxide→get_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()** |
3. Reference

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_highestValue()</code></td>
<td>Returns the maximal value observed for the CO2 concentration since the device was started.</td>
</tr>
<tr>
<td><code>get_logFrequency()</code></td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td><code>get_logicalName()</code></td>
<td>Returns the logical name of the CO2 sensor.</td>
</tr>
<tr>
<td><code>get_lowestValue()</code></td>
<td>Returns the minimal value observed for the CO2 concentration since the device was started.</td>
</tr>
<tr>
<td><code>get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>get_recordedData(startTime, endTime)</code></td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><code>get_reportFrequency()</code></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><code>get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>get_unit()</code></td>
<td>Returns the measuring unit for the CO2 concentration.</td>
</tr>
<tr>
<td><code>get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the CO2 sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the CO2 sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the CO2 sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the CO2 sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextCarbonDioxide()</code></td>
<td>Continues the enumeration of CO2 sensors started using <code>yFirstCarbonDioxide()</code>.</td>
</tr>
<tr>
<td><code>registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td></td>
</tr>
<tr>
<td>Method (Function Call)</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>carbondioxide→set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>carbondioxide→set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>carbondioxide→set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>carbondioxide→set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>carbondioxide→wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YCarbonDioxide.FindCarbonDioxide()  

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.FirstCarbonDioxide()

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.
3. Reference

**carbonDioxide->calibrateFromPoints()**

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

```java
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters :**

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

**Returns :**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
carbondioxide.describe() carbon dioxide describe()

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

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

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

carbon dioxide \rightarrow \textit{get\_advertisedValue()}
carbon dioxide \rightarrow \textit{advertisedValue()}
carbon dioxide.get\_advertisedValue()

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

\textbf{string get\_advertisedValue( )}

\begin{itemize}
  \item Returns:
    \begin{itemize}
      \item a string corresponding to the current value of the CO2 sensor (no more than 6 characters).
    \end{itemize}
  \end{itemize}

On failure, throws an exception or returns \texttt{Y\_ADVERTISEDVALUE\_INVALID}.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>carbon dioxide→get_currentRawValue()</code></td>
<td>Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.</td>
</tr>
<tr>
<td><code>carbon dioxide→currentRawValue()</code></td>
<td>Returns a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number</td>
</tr>
<tr>
<td><code>carbon dioxide.get_currentRawValue()</code></td>
<td>Returns: a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number</td>
</tr>
</tbody>
</table>

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

YCarbonDioxide
3. Reference

carbonDioxide→get_currentValue()

carbonDioxide→currentValue()

carbonDioxide.get_currentValue()

Returns the current value of the CO2 concentration, in ppm (vol), as a floating point number.

double get_currentValue()

Returns:
- a floating point number corresponding to the current value of the CO2 concentration, in ppm (vol), as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
carbon dioxide\$\rightarrow\$get\_errorMessage()

\texttt{carbon dioxide\$\rightarrow\$errorMessage()}
\texttt{carbon dioxide.get\_errorMessage()}

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

\texttt{string get\_errorMessage()}

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

\textbf{Returns}:

a string corresponding to the latest error message that occurred while using the CO2 sensor object
3. Reference

carbon dioxide → get_errorType()
carbon dioxide → errorType()
carbon dioxide.get_errorType()

YCarbonDioxide

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

YRETCODE 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
YCarbonDioxide

**参照**

`carbondioxide.get_friendlyName()`

- **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`.

---

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

```cpp
string get_friendlyName() {
    // Implementation details...
}
```
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

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

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.
YCarbonDioxide

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

```java
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`. 

```
3. Reference

```
carbondioxide→get_logFrequency()      YCarbonDioxide
carbondioxide→logFrequency()          carbon dioxide
 carpbondioxide.get_logFrequency()
```

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

```c
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`. 
**carbon dioxide**→`get_logicalName()`

**carbon dioxide**→`logicalName()`

**carbon dioxide.get_logicalName()**

Returns the logical name of the CO2 sensor.

```c
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`. 
YCarbonDioxide

carbon dioxide → get_lowestValue()
carbon dioxide → lowestValue()
carbon dioxide.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`. 

YCarbonDioxide

carbondioxide→get_module()
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 online.

Returns:

an instance of YModule
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**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 measure, 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 measure, 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.
| carbon dioxide $\rightarrow$ get_reportFrequency() | YCarbonDioxide |
| carbon dioxide $\rightarrow$ reportFrequency() |
| carbon dioxide.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`. 
<table>
<thead>
<tr>
<th>3. Reference</th>
</tr>
</thead>
</table>
| `carbon dioxide.get_resolution()` | `YCarbonDioxide`
| `carbon dioxide.resolution()` | `carbon dioxide.get_resolution()` |

Returns the resolution of the measured values.

```plaintext
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`. 
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`. 
YCarbonDioxide

YCarbonDioxide

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

Returns the value of the userData attribute, as previously stored using method set_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.
YCarbonDioxide

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

bool 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
Preloads the CO2 sensor cache with a specified validity duration.

By default, whenever accessing a device, all function attributes are kept in cache for the standard 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.
3. Reference

**carbon dioxide → load CalibrationPoints()**

YCarbonDioxide
carbon dioxide.loadCalibrationPoints()

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
continues the enumeration of CO2 sensors started using `yFirstCarbonDioxide()`.

`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.
carbon dioxide \rightarrow \text{registerTimedReportCallback()}

\text{carbon dioxide.registerTimedReportCallback()}

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

\text{int registerTimedReportCallback( TimedReportCallback \ callback)}

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

\textbf{Parameters :}

\texttt{callback} the callback 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 \texttt{YMeasure} object describing the new advertised value.
YCarbonDioxide.carbondioxide.registerValueCallback()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
carbon dioxide \rightarrow \text{set\_highest\_Value()} \quad \text{YCarbonDioxide}
carbon dioxide \rightarrow \text{set\_Highest\_Value()}
carbon dioxide.\text{set\_highest\_Value()}

Changes the recorded maximal value observed.

\begin{verbatim}
int set\_highest\_Value( double newval)
\end{verbatim}

\textbf{Parameters :}

- \texttt{newval} a floating point number corresponding to the recorded maximal value observed

\textbf{Returns :}

- \texttt{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the datalogger recording frequency for this function.

```c
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.
carbon dioxide \rightarrow \text{set\_logicalName()}
carbon dioxide \rightarrow \text{setLogicalName()}
carbon dioxide.\text{set\_logicalName()}

Changes the logical name of the CO2 sensor.

\begin{verbatim}
int \text{set\_logicalName}( \text{string \, newval})
\end{verbatim}

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

**Parameters :**

- \text{newval} a string corresponding to the logical name of the CO2 sensor.

**Returns :**

- \text{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
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.
Changes the timed value notification frequency for this function.

```c
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.
<table>
<thead>
<tr>
<th>YCarbonDioxide→set_resolution()</th>
<th>Changes the resolution of the measured physical values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>YCarbonDioxide.set_resolution()</td>
<td>The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.</td>
</tr>
</tbody>
</table>

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

```java
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. ColorLed function interface

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

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

$\text{js}$
<script type='text/javascript' src='yocto_colorled.js'></script>

$\text{nodejs}$
var yoctolib = require('yoctolib');
var YColorLed = yoctolib.YColorLed;

$\text{php}$
require_once('yocto_colorled.php');

$\text{cpp}$
#include "yocto_colorled.h"

$\text{pas}$
uses yocto_colorled;

$\text{vb}$
yocto_colorled.vb

$\text{cs}$
yocto_colorled.cs

$\text{java}$
import com.yoctopuce.YoctoAPI.YColorLed;

$\text{py}$
from yocto_colorled import *

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>yFindColorLed(func)</td>
</tr>
<tr>
<td>Retrieves an RGB led for a given identifier.</td>
</tr>
<tr>
<td>yFirstColorLed()</td>
</tr>
<tr>
<td>Starts the enumeration of RGB leds currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YColorLed methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorled→describe()</td>
</tr>
<tr>
<td>Returns a short text that describes unambiguously the instance of the RGB led in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>colorled→get_advertisedValue()</td>
</tr>
<tr>
<td>Returns the current value of the RGB led (no more than 6 characters).</td>
</tr>
<tr>
<td>colorled→get_errorMessage()</td>
</tr>
<tr>
<td>Returns the error message of the latest error with the RGB led.</td>
</tr>
<tr>
<td>colorled→get_errorType()</td>
</tr>
<tr>
<td>Returns the numerical error code of the latest error with the RGB led.</td>
</tr>
<tr>
<td>colorled→get_friendlyName()</td>
</tr>
<tr>
<td>Returns a global identifier of the RGB led in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td>colorled→get_functionDescriptor()</td>
</tr>
<tr>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td>colorled→get_functionId()</td>
</tr>
<tr>
<td>Returns the hardware identifier of the RGB led, without reference to the module.</td>
</tr>
<tr>
<td>colorled→get_hardwareId()</td>
</tr>
<tr>
<td>Returns the unique hardware identifier of the RGB led in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>colorled→get_hslColor()</td>
</tr>
<tr>
<td>Returns the current HSL color of the led.</td>
</tr>
<tr>
<td>colorled→get_logicalName()</td>
</tr>
<tr>
<td>Returns the logical name of the RGB led.</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td><code>colorled-&gt;get_module()</code></td>
</tr>
<tr>
<td><code>colorled-&gt;get_module_async(callback, context)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;get_rgbColor()</code></td>
</tr>
<tr>
<td><code>colorled-&gt;get_rgbColorAtPowerOn()</code></td>
</tr>
<tr>
<td><code>colorled-&gt;get_userData()</code></td>
</tr>
<tr>
<td><code>colorled-&gt;hslMove(hsl_target, ms_duration)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;isOnline()</code></td>
</tr>
<tr>
<td><code>colorled-&gt;isOnline_async(callback, context)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;load(msValidity)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;load_async(msValidity, callback, context)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;nextColorLed()</code></td>
</tr>
<tr>
<td><code>colorled-&gt;registerValueCallback(callback)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;set_hslColor(newval)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;set_logicalName(newval)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;set_rgbColor(newval)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;set_rgbColorAtPowerOn(newval)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;set_userData(data)</code></td>
</tr>
<tr>
<td><code>colorled-&gt;wait_async(callback, context)</code></td>
</tr>
</tbody>
</table>
YColorLed.FindColorLed() 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()

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

```csharp
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(Bad CustomeName.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`)
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.
### 3. Reference

**`colorled`** → `get_errorMessage()`  
**`colorled`** → `errorMessage()`  
**`colorled.get_errorMessage()`**

<table>
<thead>
<tr>
<th>YColorLed</th>
<th></th>
</tr>
</thead>
</table>

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

```plaintext
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
YColorLed

YRETCODE 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.
Returns a global identifier of the RGB led in the format \texttt{MODULE\_NAME.FUNCTION\_NAME}.

\texttt{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 example: \texttt{MyCustomName.relay1})

\textbf{Returns :}

- a string that uniquely identifies the RGB led using logical names (ex: \texttt{MyCustomName.relay1})

On failure, throws an exception or returns \texttt{Y\_FRIENDLYNAME\_INVALID}. 

\textbf{3. Reference}

\texttt{colorled\rightarrow get\_friendlyName()}

\texttt{colorled\rightarrow friendlyName()}

\texttt{colorled.get\_friendlyName()}

\texttt{YColorLed}
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

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

```plaintext
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`. 
YColorLed

 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.
Returns the current HSL color of the led.

```c
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`. 
Returns the logical name of the RGB led.

```cpp
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\rightarrow get\_module() 
YColorLed 

\text{colorled} \rightarrow \text{module()} \text{colorled.get\_module()}

Gets the \text{YModule} object for the device on which the function is located.

\text{YModule get\_module( )}

If the function cannot be located on any module, the returned instance of \text{YModule} is not shown as online.

\textbf{Returns :}

\begin{itemize}
  \item an instance of \text{YModule}
\end{itemize}
`colorled.get_rgbColor()`

Returns the current RGB color of the led.

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

```c
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`. 
YColorLed

```python
COLORLED

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

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

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()  

Preloads the RGB led cache with a specified validity duration.

YRETCODE load( int 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.
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.
YColorLed

```
colorled.registerValueCallback()
```

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

```
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
### YColorLed.rgbMove()

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

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

### 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.
Changes the logical name of the RGB led.

```c
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.
YColorLed

colorled→set_rgbColor()  
colorled→setRgbColor()  
colorled.set_rgbColor()

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

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

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

```c
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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YCompass = yoctolib.YCompass;
```

```php
require_once('yocto_compass.php');
```

```cpp
#include "yocto_compass.h"
```

```m
#import "yocto_compass.h"
```

```pas
uses yocto_compass;
```

```vb
yocto_compass.vb
```

```cs
yocto_compass.cs
```

```java
import com.yoctopuce.YoctoAPI.YCompass;
```

```py
from yocto_compass import *
```

### Global functions

**yFindCompass(func)**

Retrieves a compass for a given identifier.

**yFirstCompass()**

Starts the enumeration of compasses currently accessible.

### YCompass methods

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

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

**compass→describe()**

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

**compass→get_advertisedValue()**

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

**compass→get_currentRawValue()**

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

**compass→get_currentValue()**

Returns the current value of the relative bearing, in degrees, as a floating point number.

**compass→get_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 Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_highestValue()</code></td>
<td>Returns the maximal value observed for the relative bearing since the device was started.</td>
</tr>
<tr>
<td><code>get_logFrequency()</code></td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td><code>get_logicalName()</code></td>
<td>Returns the logical name of the compass.</td>
</tr>
<tr>
<td><code>get_lowestValue()</code></td>
<td>Returns the minimal value observed for the relative bearing since the device was started.</td>
</tr>
<tr>
<td><code>get_magneticHeading()</code></td>
<td>Returns the magnetic heading, regardless of the configured bearing.</td>
</tr>
<tr>
<td><code>get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>get_recordedData(startTime, endTime)</code></td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><code>get_reportFrequency()</code></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><code>get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>get_unit()</code></td>
<td>Returns the measuring unit for the relative bearing.</td>
</tr>
<tr>
<td><code>get_userData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the compass is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the compass is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the compass cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the compass cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextCompass()</code></td>
<td>Continues the enumeration of compasses started using <code>yFirstCompass()</code>.</td>
</tr>
<tr>
<td><code>registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>compass.set_logicalName(newval)</code></td>
<td>Changes the logical name of the compass.</td>
</tr>
<tr>
<td><code>compass.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>compass.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>compass.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>compass.set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>compass.wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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.
Starts the enumeration of compasses currently accessible.

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.
3. Reference

**compass → calibrateFromPoints()**

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

```java
int calibrateFromPoints(List<double> rawValues, List<double> refValues)
```

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

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

**Parameters :**

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

**Returns :**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
compass.describe() 

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

Returns :

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

YCompass

compass→get_advertisedValue()
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()**  
**compass → currentRawValue()**  
**compass.get_currentRawValue()**

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

```java
double get_currentRawValue()
```

**Returns:**

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

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`. 
Returns the current value of the relative bearing, in degrees, as a floating point number.

```java
double get_currentValue()
```

Returns:

- a floating point number corresponding to the current value of the relative bearing, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
### compass.get_errorMessage()

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

```plaintext
string get_errorMessage()
```

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

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string corresponding to the latest error message that occurred while using the compass object</td>
</tr>
</tbody>
</table>
Returns the numerical error code of the latest error with the compass.

YRETCODE 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 example: MyCustomName.relay1)

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

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.
YCompass

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

```java
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`.
}
YCompass

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

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

```java
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.
Returns the logical name of the compass.

```plaintext
string get_logicalName() 
```

**Returns:**

a string corresponding to the logical name of the compass.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
Returns the minimal value observed for the relative bearing since the device was started.

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 \texttt{Y\_LOWESTVALUE\_INVALID}.
### YCompass

**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.
YCompass

compass→get_module()
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 online.

Returns:
- an instance of YModule
YCompass

compass→get_recordedData()
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 measure, 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 measure, 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.
YCompass
compass→get_reportFrequency()
compass→reportFrequency()
compass.get_reportFrequency()

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

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.
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.
Returns the measuring unit for the relative bearing.

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

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

```text
string get_unit() {
    // Method implementation...
}
```
YCompass

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

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

```java
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.
```csharp
bool isOnline()
```

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

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()

Preloads the compass cache with a specified validity duration.

**YRETCODE load(int 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.
### YCompass

**compass.loadCalibrationPoints()**

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

```c
int loadCalibrationPoints( List<double> rawValues, List<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.
Continues the enumeration of compasses started using `yFirstCompass()`.

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()  

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

```c
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()

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

```
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
Changes the recorded maximal value observed.

```c
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()
compas→setLogFrequency()
compas.set_logFrequency()

Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the compass.

```c
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.
References

YCompass

compass\(\rightarrow\)set\_lowestValue()
compass\(\rightarrow\)setLowestValue()
compass.set\_lowestValue()

Changes the recorded minimal value observed.

\[
\text{int set\_lowestValue( double newval)}
\]

Parameters:

- \text{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.
YCompass.setReportFrequency()

Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

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

```c
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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YCurrent = yoctolib.YCurrent;
```

```php
require_once('yocto_current.php');
```

```cpp
#include "yocto_current.h"
```

```m
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, in mA, as a floating point number.

**current→get_currentValue()**

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

**current→get_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`. 
### 3. Reference

**current->get_highestValue()**
- Returns the maximal value observed for the current since the device was started.

**current->get_logFrequency()**
- Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**current->get_logicalName()**
- Returns the logical name of the current sensor.

**current->get_lowestValue()**
- Returns the minimal value observed for the current since the device was started.

**current->get_module()**
- Gets the YModule object for the device on which the function is located.

**current->get_module_async(callback, context)**
- Gets the YModule object for the device on which the function is located (asynchronous version).

**current->get_recordedData(startTime, endTime)**
- Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**current->get_reportFrequency()**
- Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**current->get_resolution()**
- Returns the resolution of the measured values.

**current->get_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.

**current->set_logFrequency(newval)**
- Changes the datalogger recording frequency for this function.

**current->set_logicalName(newval)**
- Changes the logical name of the current sensor.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>current-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>current-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>current-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>current-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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.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()

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

```cpp
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
**current.describe()**

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

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

**Returns :**

a string that describes the current sensor (ex: **Relay**(MyCustomName.relay1)**=RELAYLO1-123456.relay1**)
\begin{itemize}
\item \texttt{current \rightarrow get\_advertisedValue()}
\item \texttt{current \rightarrow advertisedValue()}
\item \texttt{current.get\_advertisedValue()}
\end{itemize}

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

\begin{Verbatim}
\textbf{string get\_advertisedValue( )}
\end{Verbatim}

\textbf{Returns :}
\begin{itemize}
\item a string corresponding to the current value of the current sensor (no more than 6 characters).
\end{itemize}

On failure, throws an exception or returns \texttt{Y\_ADVERTISEDVALUE\_INVALID}. 
current → get_currentRawValue()  
current→currentRawValue()  

`current.get_currentRawValue()`  

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

```java
double get_currentRawValue()
```

**Returns:**

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

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`. 
current→get_currentValue()
current→currentValue()current.get_currentValue()

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

```java
double get_currentValue()
```

**Returns:**

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

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
current→get_errorMessage()
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\rightarrow get\_errorType() 

current\rightarrow errorType() \ \ \ \ \ \ \ \ \ \ \ \ \ \ current.get\_errorType() 

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

YRETCODE 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\rightarrow get\_friendlyName() \quad \textbf{YCurrent}
current\rightarrow friendlyName()\quad current.get\_friendlyName()

Returns a global identifier of the current sensor in the format MODULE\_NAME.FUNCTION\_NAME.

```java
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 example: MyCustomName.relay1)

Returns:

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

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

current→get_functionDescriptor()
current→functionDescriptor()
current.get_functionDescriptor()

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

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

YFUN_DESCR get_functionDescriptor()

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
current→get_functionId()
current→functionId()current.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()  
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.
Returns the maximal value observed for the current since the device was started.

**double get_highestValue()**

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

On failure, throws an exception or returns **Y_HIGHESTVALUE_INVALID**.
### Returns

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

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`. 
current→get_logicalName()
current→logicalName()current.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.
Returns the minimal value observed for the current since the device was started.

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

On failure, throws an exception or returns \texttt{Y\_LOWESTVALUE\_INVALID}.
current\texttt{\rightarrow get\_module()}
current\texttt{\rightarrow module()}
current.get\_module()

\begin{tabular}{p{\textwidth}}
\textbf{YCurrent} \\
\end{tabular}

\textbf{Gets the YModule object for the device on which the function is located.}

\textbf{YModule get\_module( )}

If the function cannot be located on any module, the returned instance of \texttt{YModule} is not shown as online.

\textbf{Returns :}

\begin{tabular}{p{\textwidth}}
an instance of \texttt{YModule}
\end{tabular}
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 measure, 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 measure, without ending limit.

Returns:
an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.
current→get_reportFrequency()
current→reportFrequency()
current.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()  
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()
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.
Returns the value of the userData attribute, as previously stored using method `set_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()

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

**bool 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
3. Reference

current→load() current.load()

Preloads the current sensor cache with a specified validity duration.

YRETCODE load( int 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()
current.loadCalibrationPoints()

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

```c
int loadCalibrationPoints( List<double> rawValues, List<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.
<table>
<thead>
<tr>
<th>current</th>
<th>nextCurrent()</th>
<th>YCurrent</th>
</tr>
</thead>
</table>
| 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()

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

```c
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()

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

```
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the current sensor.

```c
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()

Changes the recorded minimal value observed.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code></td>
<td>a floating point number corresponding to the recorded minimal value observed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code></td>
<td>if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
current->set_reportFrequency()
current->setReportFrequency()
current.set_reportFrequency()

Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

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

```c
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. 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:

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

### Global functions

**yFindDataLogger(func)**

Retrieves a data logger for a given identifier.

**yFirstDataLogger()**

Starts the enumeration of data loggers currently accessible.

### YDataLogger methods

**datalogger→describe()**

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

**datalogger→forgetAllDataStreams()**

Clears the data logger memory and discards all recorded data streams.

**datalogger→get_advertisedValue()**

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

**datalogger→get_autoStart()**

Returns the default activation state of the data logger on power up.

**datalogger→get_beaconDriven()**

Return true if the data logger is synchronised with the localization beacon.

**datalogger→get_currentRunIndex()**

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

**datalogger→get_dataSets()**

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

**datalogger→get_dataStreams(v)**

Builds a list of all data streams hold by the data logger (legacy method).

**datalogger→get_errorMessage()**

Returns the error message of the latest error with the data logger.

**datalogger→get_errorType()**

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

**datalogger→get_friendlyName()**
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datalogger-&gt;get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_functionId()</code></td>
<td>Returns the hardware identifier of the data logger, without reference to the module.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the data logger in the form <code>SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the data logger.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_module_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_recording()</code></td>
<td>Returns the current activation state of the data logger.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_timeUTC()</code></td>
<td>Returns the Unix timestamp for current UTC time, if known.</td>
</tr>
<tr>
<td><code>datalogger-&gt;get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>datalogger-&gt;isOnline()</code></td>
<td>Checks if the data logger is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>datalogger-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the data logger is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>datalogger-&gt;load(msValidity)</code></td>
<td>Preloads the data logger cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>datalogger-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the data logger cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>datalogger-&gt;nextDataLogger()</code></td>
<td>Continues the enumeration of data loggers started using <code>yFirstDataLogger()</code>.</td>
</tr>
<tr>
<td><code>datalogger-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>datalogger-&gt;set_autoStart(newval)</code></td>
<td>Changes the default activation state of the data logger on power up.</td>
</tr>
<tr>
<td><code>datalogger-&gt;set_beaconDriven(newval)</code></td>
<td>Changes the type of synchronisation of the data logger.</td>
</tr>
<tr>
<td><code>datalogger-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the data logger.</td>
</tr>
<tr>
<td><code>datalogger-&gt;set_recording(newval)</code></td>
<td>Changes the activation state of the data logger to start/stop recording data.</td>
</tr>
<tr>
<td><code>datalogger-&gt;set_timeUTC(newval)</code></td>
<td>Changes the current UTC time reference used for recorded data.</td>
</tr>
<tr>
<td><code>datalogger-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>datalogger-&gt;wait_async(callback, context)</code></td>
<td></td>
</tr>
</tbody>
</table>
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YDataLogger.FindDataLogger()

YDataLogger.FindDataLogger(string func)

Retrieves a data logger for a given identifier.

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.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

Returns:

- a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a null pointer if there are none.

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.
YDataLogger

\texttt{describe()} \rightarrow \texttt{datalogger.describe()}

Returns a short text that describes unambiguously the instance of the data logger in the form \texttt{TYPE(NAME)=SERIAL.FUNCTIONID}.

\begin{verbatim}
string describe()

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

Returns:

A string that describes the data logger (ex: \texttt{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1})
YDataLogger

datalogger→forgetAllDataStreams()
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.
YDataLogger

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.
<table>
<thead>
<tr>
<th>YDataLogger</th>
<th>YDataLogger.get_autoStart()</th>
<th>YDataLogger.autoStart()</th>
<th>YDataLogger.get_autoStart()</th>
</tr>
</thead>
</table>

Returns the default activation state of the data logger on power up.

```plaintext
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`. 
Return true if the data logger is synchronised with the localization beacon.

`int get_beaconDriven()`

Returns:

- either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`

On failure, throws an exception or returns `Y_BEACONDRIVEN_INVALID`. 
Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

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.
Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

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 objects.

On failure, throws an exception or returns an empty list.
Builds a list of all data streams held by the data logger (legacy method).

```c
int get_dataStreams( List<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.
<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datalogger.get_errorMessage()</td>
<td>Returns the error message of the latest error with the data logger.</td>
</tr>
<tr>
<td>string get_errorMessage()</td>
<td>This method is mostly useful when using the Yoctopuce library with exceptions disabled.</td>
</tr>
</tbody>
</table>

**Returns:**
a string corresponding to the latest error message that occurred while using the data logger object
YDataLogger

dataloader\rightarrow get\_errorType()
dataloader\rightarrow errorType() datalogger.get_errorType()

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

YRETCODE 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
YDataLogger

datalogger→get_friendlyName()
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 example: MyCustomName.relay1)

Returns:

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
datalogger→get_functionDescriptor()
YDataLogger
datalogger→functionDescriptor()
datalogger.get_functionDescriptor()

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

YFUN_DESCR 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.
### YDataLogger

#### datalogger → get_functionId()

**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.
3. Reference

YDataLogger

datalogger→get_hardwareId()
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.
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()
datalogger→module() datalogger.get_module()

YDataLogger

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

`YModule get_module()`

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

**Returns:**

an instance of `YModule`
YDataLogger

dataLOGGER→get_recording()
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`. 
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datalogger.get_timeUTC()</code></td>
<td>Returns the Unix timestamp for current UTC time, if known.</td>
</tr>
<tr>
<td><code>long get_timeUTC()</code></td>
<td>Returns an integer corresponding to the Unix timestamp for current UTC time, if known. On failure, throws an exception or returns Y_TIMEUTC_INVALID.</td>
</tr>
</tbody>
</table>
YDataLogger

dataloader\rightarrow get\_userData()

dataloader\rightarrow userData()
dataloader.get\_userData()

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

Returns:
the object stored previously by the caller.

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

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

```cpp
bool 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
Preloads the data logger cache with a specified validity duration.

**YRETCODE load( int 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.
### YDataLogger

**nextDataLogger()**

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

**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.
YDataLogger

datalogger.registerValueCallback()

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

```cpp
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Changes the default activation state of the data logger on power up.

### 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.
Changes the type of synchronisation of the data logger.

```c
int set_beaconDriven( int newval)
```

Parameters:
- `newval` either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to the type of synchronisation of the data logger

Returns:
- `YAPI_SUCCESS` if the call succeeds.

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

YDataLogger
datalogger->set_logicalName()
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.
Changes the activation state of the data logger to start/stop recording data.

```c
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.
Changes the current UTC time reference used for recorded data.

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

```c
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
3.10. 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:

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

<table>
<thead>
<tr>
<th>YDataRun methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datarun.get_averageValue(measureName, pos)</code></td>
</tr>
<tr>
<td>Returns the average value of the measure observed at the specified time period.</td>
</tr>
<tr>
<td><code>datarun.get_duration()</code></td>
</tr>
<tr>
<td>Returns the duration (in seconds) of the data run.</td>
</tr>
<tr>
<td><code>datarun.get_maxValue(measureName, pos)</code></td>
</tr>
<tr>
<td>Returns the maximal value of the measure observed at the specified time period.</td>
</tr>
<tr>
<td><code>datarun.get_measureNames()</code></td>
</tr>
<tr>
<td>Returns the names of the measures recorded by the data logger.</td>
</tr>
<tr>
<td><code>datarun.get_minValue(measureName, pos)</code></td>
</tr>
<tr>
<td>Returns the minimal value of the measure observed at the specified time period.</td>
</tr>
<tr>
<td><code>datarun.get_startTimeUTC()</code></td>
</tr>
<tr>
<td>Returns the start time of the data run, relative to the Jan 1, 1970.</td>
</tr>
<tr>
<td><code>datarun.get_valueCount()</code></td>
</tr>
<tr>
<td>Returns the number of values accessible in this run, given the selected data samples interval.</td>
</tr>
<tr>
<td><code>datarun.get_valueInterval()</code></td>
</tr>
<tr>
<td>Returns the number of seconds covered by each value in this run.</td>
</tr>
<tr>
<td><code>datarun.set_valueInterval(valueInterval)</code></td>
</tr>
<tr>
<td>Changes the number of seconds covered by each value in this run.</td>
</tr>
</tbody>
</table>
YDataRun

dataRun→get_startTimeUTC()
dataRun→startTimeUTC()

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

If the UTC time was not set in the datalogger at any time during the recording of this data run, and if this is not the current run, this method returns 0.

Returns:

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data run (i.e. Unix time representation of the absolute time).
3.11. 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:

```javascript
<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"
```

```pas
uses yocto_api;
```

```vb
yocto_api.vb
```

```cs
yocto_api.cs
```

```java
import com.yoctopuce.YoctoAPI.YModule;
```

```py
from yocto_api import *
```

<table>
<thead>
<tr>
<th>YDataSet methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_endTimeUTC()</code></td>
<td>Returns the end time of the dataset, relative to the Jan 1, 1970.</td>
</tr>
<tr>
<td><code>get_functionId()</code></td>
<td>Returns the hardware identifier of the function that performed the measure, without reference to the module.</td>
</tr>
<tr>
<td><code>get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>get_measures()</code></td>
<td>Returns all measured values currently available for this DataSet, as a list of YMeasure objects.</td>
</tr>
<tr>
<td><code>get_preview()</code></td>
<td>Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.</td>
</tr>
<tr>
<td><code>get_progress()</code></td>
<td>Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.</td>
</tr>
<tr>
<td><code>get_startTimeUTC()</code></td>
<td>Returns the start time of the dataset, relative to the Jan 1, 1970.</td>
</tr>
<tr>
<td><code>get_summary()</code></td>
<td>Returns an YMeasure object which summarizes the whole DataSet.</td>
</tr>
<tr>
<td><code>get_unit()</code></td>
<td>Returns the measuring unit for the measured value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataset-&gt;loadMore()</code></td>
<td>Loads the the next block of measures from the dataLogger, and updates the progress indicator.</td>
</tr>
<tr>
<td><code>dataset-&gt;loadMore_async(callback, context)</code></td>
<td>Loads the the next block of measures from the dataLogger asynchronously.</td>
</tr>
</tbody>
</table>
dataset→get_endTimeUTC()  

YDataSet

Returns the end time of the dataset, relative to the Jan 1, 1970.

```java
long get_endTimeUTC() {
    // Implementation details here
}
```

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).
Returns the hardware identifier of the function that performed the measure, without reference to the module.

**Returns :**

A string that identifies the function (ex: temperature1)
### 3. Reference

**YDataSet**

- `dataset.get_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`. 

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_hardwareId()</td>
<td>Returns the unique hardware identifier of the function who performed the measures, in the form <code>SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td>string get_hardwareId()</td>
<td>The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example <code>THRMCPL1-123456.temperature1</code>). Returns: a string that uniquely identifies the function (ex: <code>THRMCPL1-123456.temperature1</code>). On failure, throws an exception or returns <code>Y_HARDWAREID_INVALID</code>.</td>
</tr>
</tbody>
</table>
dataset.get_measures()

YDataSet

dataset.get_measures()

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

List<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 times 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.
Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

```
List<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 instantiated by get_dataSet, the progress is zero. Each time loadMore() is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :

an integer in the range 0 to 100 (percentage of completion).
dataset→get_startTimeUTC()  YDataSet
dataset→startTimeUTC()dataset.get_startTimeUTC()

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

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).
Returns an YMeasure object which summarizes the whole DataSet.

YMeasure get_summary()

In includes the following information:
- the start of a time interval
- the end of a time interval
- the minimal value observed during the time interval
- the average value observed during the time interval
- the maximal value observed during the time interval

This summary is available as soon as loadMore() has been called for the first time.

Returns:
an YMeasure object
<table>
<thead>
<tr>
<th>YDataSet</th>
<th>dataset → get_unit()</th>
<th>dataset → unit()</th>
<th>dataset.get_unit()</th>
</tr>
</thead>
</table>

Returns the measuring unit for the measured value.

```plaintext
string get_unit()
```

**Returns**:
- a string that represents a physical unit.

On failure, throws an exception or returns `Y_UNIT_INVALID`. 
**dataset→loadMore()**

Loads the next block of measures from the dataLogger, and updates the progress indicator.

```java
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.
3.12. 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:

```<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"
```
```
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

- **get_averageValue()**: Returns the average of all measures observed within this stream.
- **get_columnCount()**: Returns the number of data columns present in this stream.
- **get_columnNames()**: Returns the title (or meaning) of each data column present in this stream.
- **get_data(row, col)**: Returns a single measure from the data stream, specified by its row and column index.
- **get_dataRows()**: Returns the whole data set contained in the stream, as a bidimensional table of numbers.
- **get_dataSamplesIntervalMs()**: Returns the number of milliseconds between two consecutive rows of this data stream.
- **get_duration()**: Returns the approximate duration of this stream, in seconds.
- **get_maxValue()**: Returns the largest measure observed within this stream.
- **get_minValue()**: Returns the smallest measure observed within this stream.
- **get_rowCount()**: Returns the number of data rows present in this stream.
- **get_runIndex()**: Returns the run index of the data stream.
- **get_startTime()**: Returns the relative start time of the data stream, measured in seconds.
- **get_startTimeUTC()**
Returns the start time of the data stream, relative to the Jan 1, 1970.
Returns the average of all measures observed within this stream.

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.
YDataStream

datastream → get_columnCount()
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.
3. Reference

dataStream→get_columnNames()
dataStream→columnNames()
dataStream.get_columnNames()

Returns the title (or meaning) of each data column present in this stream.

List<string> get_columnNames()

In most cases, 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.
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.
Returns the whole data set contained in the stream, as a bidimensional table of numbers.

List<List<double>> get_dataRows()

The meaning of the values present in each column can be obtained using the method get_columnNames().

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

**Returns:**
- A list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.
Returns the number of milliseconds between two consecutive rows of this data stream.

```c
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.
3. Reference

Returns the approximate duration of this stream, in seconds.

```c
int get_duration( )
```

**Returns:**
- the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.
datastream\rightarrow get\_maxValue() 
datastream\rightarrow maxValue() \ndatastream.get\_maxValue()

<table>
<thead>
<tr>
<th>YDataStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the largest measure observed within this stream.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>double get_maxValue()</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).</td>
</tr>
</tbody>
</table>

| On failure, throws an exception or returns Y\_DATA\_INVALID. |
YDataStream

datastream → get_minValue()
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.
Returns the number of data rows present in this stream.

```c
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.
Returns the run index of the data stream.

```cpp
int get_runIndex() {
    // Implementation
}
```

A run can be made of multiple datastreams, for different time intervals.

**Returns:**

an unsigned number corresponding to the run index.
Returns the relative start time of the data stream, measured in seconds.

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

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).
3.13. 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:

```javascript
<script type='text/javascript' src='yocto_digitalio.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YDigitalIO = yoctolib.YDigitalIO;
require_once('yocto_digitalio.php');
```
```cpp
#include "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`. 
Reference

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)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>digitalio.set_bitState(bitno, bitstate)</code></td>
<td>Changes the polarity of a single bit from the I/O port.</td>
</tr>
<tr>
<td><code>digitalio.set_logicalName(newval)</code></td>
<td>Sets a single bit of the I/O port.</td>
</tr>
<tr>
<td><code>digitalio.set_outputVoltage(newval)</code></td>
<td>Changes the logical name of the digital IO port.</td>
</tr>
<tr>
<td><code>digitalio.set_outputVoltage(newval)</code></td>
<td>Changes the voltage source used to drive output bits.</td>
</tr>
<tr>
<td><code>digitalio.set_portDirection(newval)</code></td>
<td>Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.</td>
</tr>
<tr>
<td><code>digitalio.set_portOpenDrain(newval)</code></td>
<td>Changes the electrical interface for each bit of the port.</td>
</tr>
<tr>
<td><code>digitalio.set_portPolarity(newval)</code></td>
<td>Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.</td>
</tr>
<tr>
<td><code>digitalio.set_portState(newval)</code></td>
<td>Changes the digital I/O port state: bit 0 represents input 0, and so on.</td>
</tr>
<tr>
<td><code>digitalio.set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>digitalio.toggle_bitState(bitno)</code></td>
<td>Reverts a single bit of the I/O port.</td>
</tr>
<tr>
<td><code>digitalio.wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
3. Reference

YDigitalIO.FindDigitalIO()

YDigitalIO.FindDigitalIO() 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()

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.
Schedules a pulse on a single bit for a specified duration.

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

Returns:
a string that describes the digital IO port (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
`digitalio->get_advertisedValue()`

`digitalio->advertisedValue()`

`digitalio.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`. 
Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

```c
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.
3. Reference

digitalio\rightarrow get\_bitOpenDrain()
digitalio\rightarrow bitOpenDrain() digitalio.get_bitOpenDrain()

Returns the type of electrical interface of a single bit from the I/O port.

```c
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 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.
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).

```c
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()
digitalio→bitState()
digitalio.get_bitState()

Returns the state of a single bit of the I/O port.

```c
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()
digitalio->errorMessage()
digitalio.get_errorMessage()

Returns the error message of the latest error with the digital IO port.

Returns :
a string corresponding to the latest error message that occurred while using the digital IO port object
<table>
<thead>
<tr>
<th>digitalio→get_errorType()</th>
<th>YDigitalIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>digitalio→errorType()</td>
<td>digitalio.get_errorType()</td>
</tr>
</tbody>
</table>

Returns the numerical error code of the latest error with the digital IO port.

```c
YRETCODE 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()  
digitalio -> friendlyName()  
digitalio.get_friendlyName()  

YDigitalIO

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 example: MyCustomName.relay1)

Returns:

- a string that uniquely identifies the digital IO port using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
**digitalio→get_functionDescriptor()**

**digitalio→functionDescriptor()**

digitalio.get_functionDescriptor()

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

YFUN_DESCR get_functionDescriptor()

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

**Returns :**

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
digitalio→get_functionId()
digitalio→functionId() digitalio.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()
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 RELAYL01-123456.relay1).

Returns :
   a string that uniquely identifies the digital IO port (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
Returns the logical name of the digital IO port.

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()**

**YModule get_module()**

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

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

**Returns :**

- an instance of **YModule**
digitalio->get_outputVoltage()
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()
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()
digitalio→portOpenDrain()
digitalio.get_portOpenDrain() YDigitalIO

Returns the electrical interface for each bit of the port.

```c
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()
digitalio→portPolarity()digitalio.get_portPolarity()

Returns the polarity of all the bits of the port.

```c
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\rightarrow\text{get\_portSize()}
digitalio\rightarrow\text{portSize()}
digitalio.get\_portSize()

Returns the number of bits implemented in the I/O port.

\begin{verbatim}
int \text{get\_portSize()} \\
\end{verbatim}

**Returns:**

an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns \texttt{Y\_PORTSIZE\_INVALID}.
Returns the digital IO port state: bit 0 represents input 0, and so on.

### 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`. 
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()

Checks if the digital IO port is currently reachable, without raising any error.

```cpp
bool 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()

**Preloads the digital IO port cache with a specified validity duration.**

**YRETCODE load**( int **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.
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.
Triggers a pulse on a single bit for a specified duration.

```c
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()**

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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()

Changes the direction of a single bit from the I/O port.

```c
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.
Changes the electrical interface of a single bit from the I/O port.

### 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.
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()
digitalio→setBitState()digitalio.set_bitState()

Sets a single bit of the I/O port.

```c
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.
Changes the logical name of the digital IO port.

```c
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.
Changes the voltage source used to drive output bits.

```c
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.
Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

```c
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.
Changes the electrical interface for each bit of the port.

```c
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()**

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

```cpp
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()
digitalio→setPortState()digitalio.set_portState()

Changes the digital IO port state: bit 0 represents input 0, and so on.

```c
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()
digitalio->setUserData()digitalio.set_userData()

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

```c
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() digitalio.toggle_bitState()

YDigitalIO

Reverts a single bit of the I/O port.

```c
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.
3.14. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YDisplay = yoctolib.YDisplay;
```

```php
require_once('yocto_display.php');
```

```cpp
#include "yocto_display.h"
```

```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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yFindDisplay(func)</td>
<td>Retrieves a display for a given identifier.</td>
</tr>
<tr>
<td>yFirstDisplay()</td>
<td>Starts the enumeration of displays currently accessible.</td>
</tr>
</tbody>
</table>

### YDisplay methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display→copyLayerContent(srcLayerId, dstLayerId)</td>
<td>Copies the whole content of a layer to another layer.</td>
</tr>
<tr>
<td>display→describe()</td>
<td>Returns a short text that describes unambiguously the instance of the display in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>display→fade(brightness, duration)</td>
<td>Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.</td>
</tr>
<tr>
<td>display→get_advertisedValue()</td>
<td>Returns the current value of the display (no more than 6 characters).</td>
</tr>
<tr>
<td>display→get_brightness()</td>
<td>Returns the luminosity of the module informative leds (from 0 to 100).</td>
</tr>
<tr>
<td>display→get_displayHeight()</td>
<td>Returns the display height, in pixels.</td>
</tr>
<tr>
<td>display→get_displayLayer(layerId)</td>
<td>Returns a YDisplayLayer object that can be used to draw on the specified layer.</td>
</tr>
<tr>
<td>display→get_displayType()</td>
<td>Returns the display type: monochrome, gray levels or full color.</td>
</tr>
<tr>
<td>display→get_displayWidth()</td>
<td>Returns the display width, in pixels.</td>
</tr>
<tr>
<td>display→get_enabled()</td>
<td>Returns true if the screen is powered, false otherwise.</td>
</tr>
<tr>
<td>display→get_errorMessage()</td>
<td>Returns the error message of the latest error with the display.</td>
</tr>
</tbody>
</table>
3. Reference

- **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 display 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>display.resetAll()</code></td>
<td>Clears the display screen and resets all display layers to their default state.</td>
</tr>
<tr>
<td><code>display.saveSequence(sequenceName)</code></td>
<td>Stops recording display commands and saves the sequence into the specified file on the display internal memory.</td>
</tr>
<tr>
<td><code>display.set_brightness(newval)</code></td>
<td>Changes the brightness of the display.</td>
</tr>
<tr>
<td><code>display.set_enabled(newval)</code></td>
<td>Changes the power state of the display.</td>
</tr>
<tr>
<td><code>display.set_logicalName(newval)</code></td>
<td>Changes the logical name of the display.</td>
</tr>
<tr>
<td><code>display.set_orientation(newval)</code></td>
<td>Changes the display orientation.</td>
</tr>
<tr>
<td><code>display.set_startupSeq(newval)</code></td>
<td>Changes the name of the sequence to play when the displayed is powered on.</td>
</tr>
<tr>
<td><code>display.set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>display.stopSequence()</code></td>
<td>Stops immediately any ongoing sequence replay.</td>
</tr>
<tr>
<td><code>display.swapLayerContent(layerIdA, layerIdB)</code></td>
<td>Swaps the whole content of two layers.</td>
</tr>
<tr>
<td><code>display.upload(pathname, content)</code></td>
<td>Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.</td>
</tr>
<tr>
<td><code>display.wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YDisplay.FindDisplay()  
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()

Starts the enumeration of displays currently accessible.

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() Copies the whole content of a layer to another layer.

```c
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()  

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

Returns:

a string that describes the display (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

```c
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.
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`. 
3. Reference

display\(\rightarrow\)get\_brightness()\n
display\(\rightarrow\)brightness()\ndisplay.get\_brightness()

<table>
<thead>
<tr>
<th>Returns the luminosity of the module informative leds (from 0 to 100).</th>
</tr>
</thead>
<tbody>
<tr>
<td>int get_brightness( )</td>
</tr>
</tbody>
</table>

**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.
Returns the display height, in pixels.

\[
\text{int get_displayHeight( )}
\]

**Returns:**

- an integer corresponding to the display height, in pixels

On failure, throws an exception or returns \text{Y_DISPLAYHEIGHT_INVALID}. 

\[
\text{display} \rightarrow \text{get_displayHeight()}
\]
\[
\text{display} \rightarrow \text{displayHeight()}
\]
\[
\text{display.get_displayHeight()}
\]
Returns a YDisplayLayer object that can be used to draw on the specified layer.

**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.
YDisplay

\textbf{display} \to \textbf{get\_displayType()}
\textbf{display} \to \textbf{displayType()}
display.get\_displayType()

Returns the display type: monochrome, gray levels or full color.

\begin{verbatim}
int get_displayType()
\end{verbatim}

\textbf{Returns}:

- a value among \texttt{Y\_DISPLAYTYPE\_MONO}, \texttt{Y\_DISPLAYTYPE\_GRAY} and \texttt{Y\_DISPLAYTYPE\_RGB} corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns \texttt{Y\_DISPLAYTYPE\_INVALID}. 
display→get_displayWidth()
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 \rightarrow \text{get\_enabled}() \\
Display \rightarrow \text{enabled}() \\
Display.get\_enabled() \\

Returns true if the screen is powered, false otherwise.

\textbf{int get\_enabled()} \\

\textbf{Returns :} \\
\text{either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to true if the screen is powered, false otherwise} \\

On failure, throws an exception or returns Y\_ENABLED\_INVALID.
display->get_errorMessage()
display->errorMessage(display).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
Returns the numerical error code of the latest error with the display.

YRETCODE 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
### 3. Reference

**display.get_friendlyName()**  
**YDisplay**

<table>
<thead>
<tr>
<th>String get_friendlyName()</th>
</tr>
</thead>
</table>

Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`. The returned string uses the logical names of the module and of the display if they are defined, otherwise the serial number of the module and the hardware identifier of the display (for example: `MyCustomName.relay1`).

**Returns:**
- a string that uniquely identifies the display using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
display→get_functionDescriptor()
display→functionDescriptor()
display.get_functionDescriptor()

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

YFUN_DESCR get_functionDescriptor();

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

Returns:
- an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
**display->get_functionId()**
**display->functionId()**
**display.get_functionId()**

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

```cpp
string get_functionId();
```

For example `relay1`

<table>
<thead>
<tr>
<th><strong>Returns:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>a string that identifies the display (ex: <code>relay1</code>)</td>
</tr>
</tbody>
</table>

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

YDisplay

display → get_hardwareId()
display → hardwareId()display.get_hardwareId()

<table>
<thead>
<tr>
<th>Returns</th>
<th>The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display (for example RELAYLO1-123456.relay1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>get_hardwareId()</td>
</tr>
</tbody>
</table>

**Returns :**

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

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
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()  
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.
3. Reference

**YDisplay**

\[ \text{display} \to \text{get\_layerWidth()} \]
\[ \text{display} \to \text{layerWidth()} \]
\[ \text{display.get\_layerWidth()} \]

Returns the width of the layers to draw on, in pixels.

```c
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`. 
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.
YDisplay

3. Reference

display\rightarrow get\_module()
display\rightarrow module()display.get\_module()

| YDisplay
| --- |

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\(\rightarrow\)get\_orientation()
display\(\rightarrow\)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()
display→startupSeq(display.get_startupSeq())

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string corresponding to the name of the sequence to play when the displayed is powered on</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_STARTUPSEQ_INVALID.
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()

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

```cpp
bool 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()

Preloads the display cache with a specified validity duration.

YRETCODE load( int 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()

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.
### pauseSequence

```c
int pauseSequence(int delay_ms)
```

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

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.
Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

```c
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.
### YDisplay

#### display.registerValueCallback()

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

```c
int registerValueCallback(ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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()

Clears the display screen and resets all display layers to their default state.

```cpp
int resetAll()
```

Using this function in a sequence will kill the sequence play-back. Don't use that function to reset the display at sequence start-up.

**Returns**:

- `YAPI_SUCCESS` if the call succeeds.

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

`display.saveSequence()`

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```c
int saveSequence( string sequenceName)
```

The sequence can be later replayed using `playSequence()`.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequenceName</code></td>
<td>the name of the newly created sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code></td>
<td>if the call succeeds.</td>
</tr>
<tr>
<td>On failure, throws</td>
<td>an exception or returns a negative error code.</td>
</tr>
</tbody>
</table>
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.
### YDisplay

**display->set_enabled()**

Changes the power state of the display.

```c
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.
Changes the logical name of the display.

```c
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.
Changes the display orientation.

```c
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()
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.
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()

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\text{\rightarrow}\text{swapLayerContent()}

Swaps the whole content of two layers.

\begin{verbatim}
int swapLayerContent( int layerIdA, int layerIdB)
\end{verbatim}

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 :**

- \textbf{layerIdA} the first layer (a number in range 0..layerCount-1)
- \textbf{layerIdB} the second layer (a number in range 0..layerCount-1)

**Returns :**

- \textbf{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
### display.upload() (YDisplay)

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

```c
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.
3.15. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YDisplay = yoctolib.YDisplay;
```

```php
require_once('yocto_display.php');
```

```cpp
#include "yocto_display.h"
```

```m
#include "yocto_display.h"
```

```pas
uses yocto_display;
```

```vb
requires yocto_display.vb
```

```cs
#include "yocto_display.h"
```

```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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>displaylayer→get_layerHeight()</code></td>
<td>Returns the height of the layers to draw on, in pixels.</td>
</tr>
<tr>
<td><code>displaylayer→get_layerWidth()</code></td>
<td>Returns the width of the layers to draw on, in pixels.</td>
</tr>
<tr>
<td><code>displaylayer→hide()</code></td>
<td>Hides the layer.</td>
</tr>
<tr>
<td><code>displaylayer→lineTo(x, y)</code></td>
<td>Draws a line from current drawing pointer position to the specified position.</td>
</tr>
<tr>
<td><code>displaylayer→moveTo(x, y)</code></td>
<td>Moves the drawing pointer of this layer to the specified position.</td>
</tr>
<tr>
<td><code>displaylayer→reset()</code></td>
<td>Reverts the layer to its initial state (fully transparent, default settings).</td>
</tr>
<tr>
<td><code>displaylayer→selectColorPen(color)</code></td>
<td>Selects the pen color for all subsequent drawing functions, including text drawing.</td>
</tr>
<tr>
<td><code>displaylayer→selectEraser()</code></td>
<td>Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.</td>
</tr>
<tr>
<td><code>displaylayer→selectFont(fontname)</code></td>
<td>Selects a font to use for the next text drawing functions, by providing the name of the font file.</td>
</tr>
<tr>
<td><code>displaylayer→selectGrayPen(graylevel)</code></td>
<td>Selects the pen gray level for all subsequent drawing functions, including text drawing.</td>
</tr>
<tr>
<td><code>displaylayer→setAntialiasingMode(mode)</code></td>
<td>Enables or disables anti-aliasing for drawing oblique lines and circles.</td>
</tr>
<tr>
<td><code>displaylayer→setConsoleBackground(bgcol)</code></td>
<td>Sets up the background color used by the <code>clearConsole</code> function and by the console scrolling feature.</td>
</tr>
<tr>
<td><code>displaylayer→setConsoleMargins(x1, y1, x2, y2)</code></td>
<td>Sets up display margins for the <code>consoleOut</code> function.</td>
</tr>
<tr>
<td><code>displaylayer→setConsoleWordWrap(wordwrap)</code></td>
<td>Sets up the wrapping behaviour used by the <code>consoleOut</code> function.</td>
</tr>
<tr>
<td><code>displaylayer→setLayerPosition(x, y, scrollTime)</code></td>
<td>Sets the position of the layer relative to the display upper left corner.</td>
</tr>
<tr>
<td><code>displaylayer→unhide()</code></td>
<td>Shows the layer.</td>
</tr>
</tbody>
</table>
displaylayer→clear() displaylayer.clear()

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()  
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\rightarrow\text{consoleOut()}$

\text{consoleOut()}$

Outputs a message in the console area, and advances the console pointer accordingly.

\begin{verbatim}
int consoleOut( string text)
\end{verbatim}

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\rightarrow \text{drawBar()}

\begin{tabular}{ll}
\text{drawBar()} & \text{YDisplayLayer} \\
\end{tabular}

Draws a filled rectangular bar at a specified position.

\begin{tabular}{ll}
\text{int drawBar( int } x1, \text{ int } y1, \text{ int } x2, \text{ int } y2) \\
\end{tabular}

\textbf{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

\textbf{Returns :}
- \text{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
Draws a bitmap at the specified position.

```c
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() \hspace{1cm} \text{drawCircle()} \hspace{1cm} YDisplayLayer

Draws an empty circle at a specified position.

\begin{verbatim}
int drawCircle(int x, int y, int r)
\end{verbatim}

**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:**
- \texttt{YAPI\_SUCCESS} if the call succeeds.

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

**displaylayer**→**drawDisc()**

displaylayer.drawDisc()

YDisplayLayer

Draws a filled disc at a given position.

<table>
<thead>
<tr>
<th>int drawDisc( int x, int y, int r)</th>
</tr>
</thead>
</table>

**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**

Draws a GIF image at the specified position.

### 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.

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.
3. Reference

抵抗层 drawPixel() drawLayer.drawPixel() YDisplayLayer

Draws a single pixel at the specified position.

```c
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()**  
Draws an empty rectangle at a specified position.

```c
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.
3. Reference

displaylayer → drawText() displaylayer.drawText()

YDisplayLayer

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()`  

Gets parent YDisplay.

**YDisplay get_display()**

Returns the parent YDisplay object of the current YDisplayLayer.

**Returns:**

an YDisplay object
YDisplayLayer

displaylayer.get_displayHeight()

displaylayer → displayHeight()

displaylayer → get_displayHeight()

<table>
<thead>
<tr>
<th>3. Reference</th>
</tr>
</thead>
</table>

Returns the display height, in pixels.

```c
int get_displayHeight()
```

**Returns:**

- an integer corresponding to the display height, in pixels
- On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.
displaylayer\rightarrow \textit{get\_displayWidth}() \hspace{2cm} \textbf{YDisplayLayer}
displaylayer\rightarrow \textit{displayWidth}() \hspace{2cm} \textbf{displaylayer.get\_displayWidth}()

\begin{center}
\begin{tabular}{|l|}
\hline
Returns the display width, in pixels. \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{int get\_displayWidth( )} \\
\hline
Returns : \hspace{2cm} \\
an integer corresponding to the display width, in pixels On failure, throws an exception or returns \textbf{Y\_DISPLAYWIDTH\_INVALID}. \\
\end{tabular}
\end{center}
displaylayer→get_layerHeight()

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()
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()

Hides the layer.

int hide()

The state of the layer is preserved 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() 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()**

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()

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.
selectColorPen() selects the pen color for all subsequent drawing functions, including text drawing.

```c
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()

Selects an eraser instead of a pen for all subsequent drawing functions, except for 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.
selectFont

YDisplayLayer

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```cpp
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()**

Selects the pen gray level for all subsequent drawing functions, including text drawing.

```c
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() Enables or disables anti-aliasing for drawing oblique lines and circles.

```c
int setAntialiasingMode( bool mode)
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

**Parameters:**
- `mode` true to enable antialiasing, false to disable it.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
**displaylayer→setConsoleBackground()**  
**displaylayer.setConsoleBackground()**

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.
3. Reference

displaylayer.setConsoleMargins()

displaylayer.setConsoleMargins()

Sets up display margins for the consoleOut function.

```c
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()
displaylayer.setConsoleWordWrap()

Sets up the wrapping behaviour used by the consoleOut function.

int setConsoleWordWrap( bool wordwrap)

**Parameters:**

- `wordwrap` true to wrap only between words, false to wrap on the last column anyway.

**Returns:**

- YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
displaylayer→setLayerPosition()
displaylayer.setLayerPosition()

Sets the position of the layer relative to the display upper left corner.

```c
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.
<table>
<thead>
<tr>
<th>displaylayer</th>
<th>→ unhide()</th>
<th>displaylayer.unhide()</th>
<th>YDisplayLayer</th>
</tr>
</thead>
</table>

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.
3.16. 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:

```javascript
<script type='text/javascript' src='yocto_dualpower.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YDualPower = yoctolib.YDualPower;
```
```php
require_once('yocto_dualpower.php');
```
```cpp
#include "yocto_dualpower.h"
```
```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()**
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>get_powerControl()</code></td>
<td>Returns the selected power source for module functions that require lots of current.</td>
</tr>
<tr>
<td><code>get_powerState()</code></td>
<td>Returns the current power source for module functions that require lots of current.</td>
</tr>
<tr>
<td><code>get_userData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the power control is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the power control is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the power control cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the power control cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextDualPower()</code></td>
<td>Continues the enumeration of dual power controls started using <code>yFirstDualPower()</code>.</td>
</tr>
<tr>
<td><code>registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td>Changes the logical name of the power control.</td>
</tr>
<tr>
<td><code>set_powerControl(newval)</code></td>
<td>Changes the selected power source for module functions that require lots of current.</td>
</tr>
<tr>
<td><code>set_userData(data)</code></td>
<td>Stores a user context provided as argument in the <code>userData</code> attribute of the function.</td>
</tr>
<tr>
<td><code>wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
Retrieves a dual power control for a given identifier.

YDualPower\texttt{ FindDualPower( string }\texttt{ func})

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 \texttt{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.

\begin{itemize}
  \item \textbf{Parameters :}
    \begin{itemize}
      \item \texttt{func} a string that uniquely characterizes the power control
    \end{itemize}
  \item \textbf{Returns :}
    \begin{itemize}
      \item a \texttt{YDualPower} object allowing you to drive the power control.
    \end{itemize}
\end{itemize}
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()

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

string describe()

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

Returns :
- a string that describes the power control (ex: Relay(MyCustomName.relay1)=RELAYLO1-
  123456.relay1)
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dualpower.get_advertisedValue()</td>
<td>Returns the current value of the power control (no more than 6 characters).</td>
</tr>
<tr>
<td>string get_advertisedValue()</td>
<td>Returns:</td>
</tr>
<tr>
<td></td>
<td>a string corresponding to the current value of the power control (no more than 6 characters).</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.</td>
</tr>
</tbody>
</table>
dualpower \rightarrow \text{get\_errorMessage}() \\
\text{dualpower} \rightarrow \text{errorMessage}() \\
\text{dualpower.get\_errorMessage}() \\

Returns the error message of the latest error with the power control.

\begin{verbatim}
string \text{get\_errorMessage}()
\end{verbatim}

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

\textbf{Returns :} \\
a string corresponding to the latest error message that occurred while using the power control object
dualpower→get_errorType() YDualPower
dualpower→errorType() dualpower.get_errorType()

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

YRETCODE 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()
dualpower→extVoltage() dualpower.get_extVoltage()

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

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.
### 3. Reference

**dualpower→get_friendlyName()**

**dualpower→friendlyName()**

**dualpower.get_friendlyName()**

YDualPower

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

```java
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`. 

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

<table>
<thead>
<tr>
<th>YFUN_DESCR</th>
<th>get_functionDescriptor()</th>
</tr>
</thead>
</table>

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`.*
YDualPower

dualpower→get_functionId()
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()
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()  
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 online.

**Returns:**

- an instance of YModule
Returns the selected power source for module functions that require lots of current.

```c
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 \rightarrow \text{get\_powerState()}

dualpower \rightarrow \text{powerState()}

dualpower.get\_powerState()

Returns the current power source for module functions that require lots of current.

\begin{verbatim}
int \text{get\_powerState()}
\end{verbatim}

\textbf{Returns :}

A value among \texttt{Y\_POWERSTATE\_OFF}, \texttt{Y\_POWERSTATE\_FROM\_USB} and \texttt{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 \texttt{Y\_POWERSTATE\_INVALID}.
Returns the value of the userData attribute, as previously stored using method `set_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()

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

```plaintext
bool 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
By default, whenever accessing a device, all function attributes are kept in cache for the standard 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.
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()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
Changes the logical name of the power control.

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.
Changes the selected power source for module functions that require lots of current.

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

```c
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
### 3.17. 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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>js</td>
<td><code>&lt;script type='text/javascript' src='yocto_files.js'&gt;&lt;/script&gt;</code></td>
</tr>
<tr>
<td>nodejs</td>
<td><code>var yoctolib = require('yoctolib');</code>&lt;br&gt;<code>var YFiles = yoctolib.YFiles;</code></td>
</tr>
<tr>
<td>php</td>
<td><code>require_once('yocto_files.php');</code>&lt;br&gt;<code>#include &quot;yocto_files.h&quot;</code>&lt;br&gt;<code>#import &quot;yocto_files.h&quot;</code>&lt;br&gt;<code>uses yocto_files;</code>&lt;br&gt;<code>yocto_files.vb</code>&lt;br&gt;<code>yocto_files.cs</code>&lt;br&gt;<code>pull yocto_files.cs</code>&lt;br&gt;<code>java</code>&lt;br&gt;<code>import com.yoctopuce.YoctoAPI.YFiles;</code>&lt;br&gt;<code>from yocto_files import *</code></td>
</tr>
<tr>
<td>cpp</td>
<td><code>#include &quot;yocto_files.h&quot;</code></td>
</tr>
<tr>
<td>pas</td>
<td><code>uses yocto_files;</code></td>
</tr>
<tr>
<td>vb</td>
<td><code>yocto_files.vb</code></td>
</tr>
<tr>
<td>cs</td>
<td><code>yocto_files.cs</code></td>
</tr>
<tr>
<td>java</td>
<td><code>import com.yoctopuce.YoctoAPI.YFiles;</code></td>
</tr>
<tr>
<td>py</td>
<td><code>from yocto_files import *</code></td>
</tr>
</tbody>
</table>

**Global functions**

- **yFindFiles(func)**<br>Retrieves a filesystem for a given identifier.

- **yFirstFiles()**<br>Starts the enumeration of filesystems currently accessible.

**YFiles methods**

- **files→describe()**<br>Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **files→download(pathname)**<br>Downloads the requested file and returns a binary buffer with its content.

- **files→download_async(pathname, callback, context)**<br>Downloads the requested file and returns a binary buffer with its content.

- **files→format_fs()**<br>Reinitialize the filesystem to its clean, unfragmented, empty state.

- **files→get_advertisedValue()**<br>Returns the current value of the filesystem (no more than 6 characters).

- **files→get_errorMessage()**<br>Returns the error message of the latest error with the filesystem.

- **files→get_errorType()**<br>Returns the numerical error code of the latest error with the filesystem.

- **files→get_filesCount()**<br>Returns the number of files currently loaded in the filesystem.

- **files→get_freeSpace()**<br>Returns the free space for uploading new files to the filesystem, in bytes.

- **files→get_friendlyName()**<br>Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

- **files→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **files→get_functionId()**
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>files-&gt;get_hardwareId()</code></td>
<td>Returns the hardware identifier of the filesystem, without reference to the module.</td>
</tr>
<tr>
<td><code>files-&gt;get_list(pattern)</code></td>
<td>Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.</td>
</tr>
<tr>
<td><code>files-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the filesystem.</td>
</tr>
<tr>
<td><code>files-&gt;get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>files-&gt;get_module_async(callback, context)</code></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>files-&gt;get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>files-&gt;isOnline()</code></td>
<td>Checks if the filesystem is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>files-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the filesystem is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>files-&gt;load(msValidity)</code></td>
<td>Preloads the filesystem cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>files-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the filesystem cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>files-&gt;nextFiles()</code></td>
<td>Continues the enumeration of filesystems started using <code>yFirstFiles()</code></td>
</tr>
<tr>
<td><code>files-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>files-&gt;remove(pathname)</code></td>
<td>Deletes a file, given by its full path name, from the filesystem.</td>
</tr>
<tr>
<td><code>files-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the filesystem.</td>
</tr>
<tr>
<td><code>files-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>files-&gt;upload(pathname, content)</code></td>
<td>Uploads a file to the filesystem, to the specified full path name.</td>
</tr>
<tr>
<td><code>files-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YFiles.FindFiles() 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.FirstFiles()**

Starts the enumeration of filesystems currently accessible.

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

string describe( )

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

Returns :
   a string that describes the filesystem (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
files->format_fs() files.format_fs()

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

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

```cpp
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()
files → errorType() files.get_errorType()

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

YRETCODE 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
3. Reference

Returns the number of files currently loaded in the filesystem.

```cpp
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()
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.
3. Reference

**files.get_friendlyName()**

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

```plaintext
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 example: `MyCustomName.relay1`)

**Returns:**
- a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
Returns a unique identifier of type \texttt{YFUN\_DESCR} corresponding to the function.

\begin{verbatim}
YFUN\_DESCR get\_functionDescriptor() 
\end{verbatim}

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

\textbf{Returns :}

an identifier of type \texttt{YFUN\_DESCR}.

\textit{If the function has never been contacted, the returned value is \texttt{Y\_FUNCTIONDESCRIPTOR\_INVALID}.}
Returns the hardware identifier of the filesystem, without reference to the module.

```csharp
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`. 
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.
Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

List&lt;YFileRecord&gt; 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.
Returns the logical name of the filesystem.

```java
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()`

**YModule get_module()**

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

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

**Returns:**

- an instance of `YModule`
### YFiles

files → `get_userData()`

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.

<table>
<thead>
<tr>
<th>Returns</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>the object stored previously by the caller.</td>
<td></td>
</tr>
</tbody>
</table>
3. Reference

YFiles

YFiles.files.isFileOnline()

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

```csharp
bool 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() \( \text{files.load()} \)

Preloads the filesystem cache with a specified validity duration.

**YRETCODE load( int 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()  

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()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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() \texttt{files.remove()}

Deletes a file, given by its full path name, from the filesystem.

\begin{verbatim}
int remove( string pathname)
\end{verbatim}

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 \texttt{format_fs} to fully reinitialize the filesystem.

**Parameters :**
- \texttt{pathname} path and name of the file to remove.

**Returns :**
- \texttt{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
files.set_logicalName()

Changes the logical name of the filesystem.

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

```java
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()

Uploads a file to the filesystem, to the specified full path name.

```c
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. Reference

3.18. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YGenericSensor = yoctolib.YGenericSensor;
```

```php
require_once('yocto_genericsensor.php');
```

```cpp
#include "yocto_genericsensor.h"
```

```m```
```
```

```pa```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_highestValue()</code></td>
<td>Returns the maximal value observed for the measure since the device was started.</td>
</tr>
<tr>
<td><code>get_logFrequency()</code></td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td><code>get_logicalName()</code></td>
<td>Returns the logical name of the generic sensor.</td>
</tr>
<tr>
<td><code>get_lowestValue()</code></td>
<td>Returns the minimal value observed for the measure since the device was started.</td>
</tr>
<tr>
<td><code>get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>get_recordedData(startTime, endTime)</code></td>
<td>Retrieves a <code>DataSet</code> object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><code>get_reportFrequency()</code></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><code>get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>get_signalBias()</code></td>
<td>Returns the electric signal bias for zero shift adjustment.</td>
</tr>
<tr>
<td><code>get_signalRange()</code></td>
<td>Returns the electric signal range used by the sensor.</td>
</tr>
<tr>
<td><code>get_signalUnit()</code></td>
<td>Returns the measuring unit of the electrical signal used by the sensor.</td>
</tr>
<tr>
<td><code>get_signalValue()</code></td>
<td>Returns the measured value of the electrical signal used by the sensor.</td>
</tr>
<tr>
<td><code>get_unit()</code></td>
<td>Returns the measuring unit for the measure.</td>
</tr>
<tr>
<td><code>get_userData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>get_valueRange()</code></td>
<td>Returns the physical value range measured by the sensor.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the generic sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the generic sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the generic sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextGenericSensor()</code></td>
<td>Continues the enumeration of generic sensors started using <code>yFirstGenericSensor()</code>.</td>
</tr>
</tbody>
</table>
3. Reference

**genericsensor** → **registerTimedReportCallback** *(callback)*
Registers the callback function that is invoked on every periodic timed notification.

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

**genericsensor** → **set_highestValue** *(newval)*
Changes the recorded maximal value observed.

**genericsensor** → **set_logFrequency** *(newval)*
Changes the datalogger recording frequency for this function.

**genericsensor** → **set_logicalName** *(newval)*
Changes the logical name of the generic sensor.

**genericsensor** → **set_lowestValue** *(newval)*
Changes the recorded minimal value observed.

**genericsensor** → **set_reportFrequency** *(newval)*
Changes the timed value notification frequency for this function.

**genericsensor** → **set_resolution** *(newval)*
Changes the resolution of the measured physical values.

**genericsensor** → **set_signalBias** *(newval)*
Changes the electric signal bias for zero shift adjustment.

**genericsensor** → **set_signalRange** *(newval)*
Changes the electric signal range used by the sensor.

**genericsensor** → **set_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.

**genericsensor** → **zeroAdjust()**
Adjusts the signal bias so that the current signal value is need precisely as zero.
YGenericSensor.FindGenericSensor()  YGenericSensor

YGenericSensor.FindGenericSensor()
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.FirstGenericSensor()

YGenericSensor.FirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

Use the method YGenericSensor.nextGenericSensor() to iterate on next generic sensors.

Returns:

a pointer to a YGenericSensor object, corresponding to the first generic sensor currently online, or a null pointer if there are none.
genericsensor→calibrateFromPoints()  
genericsensor.calibrateFromPoints()

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

```c
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters :**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
Genericsensor→describe()\hspace{1em}\text{genericsensor.describe()}

YGenericSensor

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

```plaintext
string describe()
```

More precisely, `TYPE` is the type of the function, `NAME` the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns

```
Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1
```

if the module is already connected or

```
Relay(BadCustomeName.relay1)=unresolved
```

if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:

a string that describes the generic sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_advertisedValue()</code></td>
<td>Returns the current value of the generic sensor (no more than 6 characters).</td>
</tr>
<tr>
<td><code>advertisedValue()</code></td>
<td>Returns:</td>
</tr>
<tr>
<td></td>
<td>a string corresponding to the current value of the generic sensor (no more than 6 characters).</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.</td>
</tr>
</tbody>
</table>
Returns the uncalibrated, unrounded raw value returned by the sensor.

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

On failure, throws an exception or returns \texttt{Y_CURRENTRAWVALUE_INVALID}.
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`. 
Returns the error message of the latest error with the generic sensor.

```cpp
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
**getYetCode**

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

**Returns:**

a number corresponding to the code of the latest error that occurred while using the generic sensor object.
YGenericSensor

genericsensor→get_friendlyName()

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.

YFUN_DESCR 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.
YGenericSensor

genericsensor.get_functionId()  
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()
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.
3. Reference

`genericsensor.get_highestValue()`
`YGenericSensor`
`genericsensor.highestValue()`
`genericsensor.get_highestValue()`

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

```java
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.
YGenericSensor → get_logicalName()  
YGenericSensor → logicalName()  
YGenericSensor.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.
### 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**.
3. Reference

```python
YGenericSensor

genericsensor.get_module()  
genericsensor.module()  
genericsensor.get_module()
```

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

```python
YModule get_module()
```

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

**Returns:**

- an instance of `YModule`
Retrieves a Data\textit{S}et object holding historical data for this sensor, for a specified time interval.

\begin{verbatim}
YDataSet get_recordedData( long startTime, long endTime)
\end{verbatim}

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the Data\textit{S}et 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 Data\textit{S}et 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 measure, 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 measure, without ending limit.

**Returns :**

an instance of YData\textit{S}et, providing access to historical data. Past measures can be loaded progressively using methods from the YData\textit{S}et object.
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.

```c
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`. 
YGenericSensor
genericsensor→get_signalBias()
genericsensor→signalBias()
genericsensor.get_signalBias()

Returns the electric signal bias for zero shift adjustment.

double get_signalBias()

A positive bias means that the signal is over-reporting the measure, while a negative bias means that the signal is underreporting the measure.

Returns:
- a floating point number corresponding to the electric signal bias for zero shift adjustment

On failure, throws an exception or returns Y_SIGNALBIAS_INVALID.
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.
### 3. Reference

<table>
<thead>
<tr>
<th>YGenericSensor</th>
<th>genericsensor.get_signalUnit()</th>
</tr>
</thead>
<tbody>
<tr>
<td>genericsensor</td>
<td>signalUnit()</td>
</tr>
<tr>
<td>genericsensor</td>
<td>get_signalUnit()</td>
</tr>
</tbody>
</table>

Returns the measuring unit of the electrical signal used by the sensor.

```cpp
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`. 
**get_signalValue()**

Returns the measured value of the electrical signal used by the sensor.

```java
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**.
### 3. Reference

<table>
<thead>
<tr>
<th>genericsensor → get_unit()</th>
<th>YGenericSensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>genericsensor → unit()</td>
<td>genericsensor.get_unit()</td>
</tr>
</tbody>
</table>

Returns the measuring unit for the measure.

```plaintext
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()
Genericsensor→userData()
Genericsensor.get_userData()

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

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.
Returns the physical value range measured by the sensor.

```cpp
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()genericsensor.isOnline()  
YGenericSensor

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

bool 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
YGenericSensor

Preloads the generic sensor cache with a specified validity duration.

\[ \text{YRETCODE load( int msValidity)} \]

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.
YGenericSensor

genericsensor->loadCalibrationPoints()

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nextGenericSensor()</code></td>
<td>Continues the enumeration of generic sensors started using <code>yFirstGenericSensor()</code>.</td>
</tr>
</tbody>
</table>

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

`int registerValueCallback( ValueCallback callback)`

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

**Parameters:**
- `callback` the callback 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.
Changes the recorded maximal value observed.

**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.
Changes the datalogger recording frequency for this function.

```cpp
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.
Changes the logical name of the generic sensor.

```c
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.
Changes the recorded minimal value observed.

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.
3. Reference

YGenericSensor
genericsensor→set_reportFrequency()
genericsensor→setReportFrequency()
genericsensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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\textrightarrow\texttt{set\_signalBias()}

Changes the electric signal bias for zero shift adjustment.

\textbf{Parameters :}

\texttt{newval} a floating point number corresponding to the electric signal bias for zero shift adjustment

\textbf{Returns :}

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the electric signal range used by the sensor.

```c
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.
changes the measuring unit for the measured value.

```c
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()
Genericsensor → setUserData()
Genericsensor.set_userData()

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

```c
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
Changes the physical value range measured by the sensor.

```c
int setValueRange( string newval)
```

As a side effect, the range modification may automatically modify the display resolution.

**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. Reference

genericsensor→zeroAdjust()
genericsensor.zeroAdjust()

Adjusts the signal bias so that the current signal value is need precisely as zero.

Returns:

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
3.19. 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:

```javascript
<script type='text/javascript' src='yocto_gyro.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YGyro = yoctolib.YGyro;
require_once('yocto_gyro.php');
```
```cpp
#include "yocto_gyro.h"
```
```m
#include "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, in degrees per second, as a floating point number.

**gyro→get_currentValue()**

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

**gyro→get_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()**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gyro→get_heading()</td>
<td>Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.</td>
</tr>
<tr>
<td>gyro→get_highestValue()</td>
<td>Returns the maximal value observed for the angular velocity since the device was started.</td>
</tr>
<tr>
<td>gyro→get_logFrequency()</td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td>gyro→get_logicalName()</td>
<td>Returns the logical name of the gyroscope.</td>
</tr>
<tr>
<td>gyro→get_lowestValue()</td>
<td>Returns the minimal value observed for the angular velocity since the device was started.</td>
</tr>
<tr>
<td>gyro→get_module()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>gyro→get_module_async(callback, context)</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>gyro→get_pitch()</td>
<td>Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.</td>
</tr>
<tr>
<td>gyro→get_quaternionW()</td>
<td>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.</td>
</tr>
<tr>
<td>gyro→get_quaternionX()</td>
<td>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.</td>
</tr>
<tr>
<td>gyro→get_quaternionY()</td>
<td>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.</td>
</tr>
<tr>
<td>gyro→get_quaternionZ()</td>
<td>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.</td>
</tr>
<tr>
<td>gyro→get_recordedData(startTime, endTime)</td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td>gyro→get_reportFrequency()</td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td>gyro→get_resolution()</td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td>gyro→get_roll()</td>
<td>Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.</td>
</tr>
<tr>
<td>gyro→get_unit()</td>
<td>Returns the measuring unit for the angular velocity.</td>
</tr>
<tr>
<td>gyro→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>gyro→get_xValue()</td>
<td></td>
</tr>
</tbody>
</table>
Returns the angular velocity around the X axis of the device, as a floating point number.

```
gyro→get_yValue()
```

Returns the angular velocity around the Y axis of the device, as a floating point number.

```
gyro→get_zValue()
```

Returns the angular velocity around the Z axis of the device, as a floating point number.

```
gyro→isOnline()
```

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

```
gyro→isOnline_async(callback, context)
```

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

```
gyro→load(msValidity)
```

Preloads the gyroscope cache with a specified validity duration.

```
gyro→loadCalibrationPoints(rawValues, refValues)
```

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

```
gyro→load_async(msValidity, callback, context)
```

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

```
gyro→nextGyro()
```

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

```
gyro→registerAnglesCallback(callback)
```

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
gyro→registerQuaternionCallback(callback)
```

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
gyro→registerTimedReportCallback(callback)
```

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

```
gyro→registerValueCallback(callback)
```

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

```
gyro→set_highestValue(newval)
```

Changes the recorded maximal value observed.

```
gyro→set_logFrequency(newval)
```

Changes the datalogger recording frequency for this function.

```
gyro→set_logicalName(newval)
```

Changes the logical name of the gyroscope.

```
gyro→set_lowestValue(newval)
```

Changes the recorded minimal value observed.

```
gyro→set_reportFrequency(newval)
```

Changes the timed value notification frequency for this function.

```
gyro→set_resolution(newval)
```

Changes the resolution of the measured physical values.

```
gyro→set_userData(data)
```

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

```
gyro→wait_async(callback, context)
```

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

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()
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.
3. Reference

**gyro.calibrateFromPoints()**

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

```java
int calibrateFromPoints(List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
gyro.describe() \[\text{YGyro}\]

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

```
string describe()
```

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

**Returns:**

A string that describes the gyroscope (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
Returns the current value of the gyroscope (no more than 6 characters).

```java
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_ADVERTISED_VALUE_INVALID`.
Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.

double get_currentRawValue()

**Returns:**

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
Returns the current value of the angular velocity, in degrees per second, as a floating point number.

Returns:
- a floating point number corresponding to the current value of the angular velocity, in degrees per second, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
**get(errorMessage)**

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

```java
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.
Returns the numerical error code of the latest error with the gyroscope.

**YGyro**

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

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

Returns:
- A string that uniquely identifies the gyroscope using logical names (ex: `MyCustomName.relay1`)

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

```
gyro → get_friendlyName()
gyro → friendlyName() gyro.get_friendlyName()```
Returns a unique identifier of type \texttt{YFUN\_DESCR} corresponding to the function.

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

\textbf{Returns :}

- an identifier of type \texttt{YFUN\_DESCR}.

\textit{If the function has never been contacted, the returned value is \texttt{Y\_FUNCTIONDESCRIPTOR\_INVALID}.}
gyro\rightarrow \text{get\_functionId()}
gyro\rightarrow \text{functionId()}
gyro.get\_functionId()

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

\text{string get\_functionId( )}

For example relay1

\textbf{Returns}:

a string that identifies the gyroscope (ex: relay1)

On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.
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.
```cpp
// 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.
```
3. Reference

$\text{gyro} \rightarrow \text{get\_highestValue()}$

$\text{gyro} \rightarrow \text{highestValue()}$

$\text{gyro}.\text{get\_highestValue()}$

Returns the maximal value observed for the angular velocity since the device was started.

\[
\text{double get\_highestValue()} \\
\]

**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()”
“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.
3. Reference

 yg → get_logicalName()
 yg → logicalName()
 yg.get_logicalName()

 Returns the logical name of the gyroscope.

<table>
<thead>
<tr>
<th>string get_logicalName( )</th>
</tr>
</thead>
</table>

**Returns:**
- A string corresponding to the logical name of the gyroscope.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
gyro→get_lowestValue()
gyro→lowestValue() gyro.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.
3. Reference

**YGyro**

`gyro->get_module()`  
`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 online.

**Returns:**
- an instance of `YModule`
**gyro→get_pitch()**

**gyro→pitch() gyro.get_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```plaintext
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.
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.
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.
3. Reference

YGyro

gyro → get_quaternionY()

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.
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.

**Returns:**

A floating-point number corresponding to the $z$ component of the quaternion.
3. Reference

$\text{gyro} \rightarrow \text{get\_recordedData()}$

$\text{gyro} \rightarrow \text{recordedData()}$

$\text{gyro}.\text{get\_recordedData()}$

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

**YGyro**

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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startTime</strong></td>
<td>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 measure, without initial limit.</td>
</tr>
<tr>
<td><strong>endTime</strong></td>
<td>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 measure, without ending limit.</td>
</tr>
</tbody>
</table>

<p>| Returns             | an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object. |</p>
<table>
<thead>
<tr>
<th>gyro→get_reportFrequency()</th>
<th>YGyro gyro→reportFrequency()gyro.get_reportFrequency()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
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.
Returns the resolution of the measured values.

```c
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.
YGyro

3. Reference

**YGyro**

```c
get_roll()
```

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```c
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.
3. Reference

YGyro

\[
\text{gyro} \rightarrow \text{get\_unit()}
\]

\[
\text{gyro} \rightarrow \text{unit()} \text{gyro.get\_unit()}
\]

Returns the measuring unit for the angular velocity.

\[
\text{string get\_unit()}
\]

**Returns:**

A string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns Y\_UNIT\_INVALID.
Returns the value of the userData attribute, as previously stored using method `set_userData`.  

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

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>the object stored previously by the caller.</td>
</tr>
</tbody>
</table>
### YGyro

**gyro.get_xValue()**  
**Returns:**  
Returns the angular velocity around the X axis of the device, as a floating point number.  

```java
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**.
Returns the angular velocity around the Y axis of the device, as a floating point number.

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_VALUE_INVALID.
### 3. Reference

<table>
<thead>
<tr>
<th>gyro→get_zValue()</th>
<th>YGyro</th>
</tr>
</thead>
<tbody>
<tr>
<td>gyro→zValue()</td>
<td>gyro.get_zValue()</td>
</tr>
</tbody>
</table>

Returns the angular velocity around the Z axis of the device, as a floating point number.

```c
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.
}
```
Y Gyro

```python
bool isOnline()
```

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

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.
By default, whenever accessing a device, all function attributes are kept in cache for the standard 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()**

**gyro.loadCalibrationPoints()**

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
3. Reference

**GYro** → **nextGyro()**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

**Y Gyro nextGyro()**

**Returns:**

a pointer to a **Y Gyro** object, corresponding to a gyroscope currently online, or a null pointer if there are no more gyroscopes to enumerate.
Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```c
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).
3. Reference

**gyro@registerQuaternionCallback()**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```c
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).
3. Reference

**gyro**\rightarrow\texttt{registerTimedReportCallback()}

**gyro.registerTimedReportCallback()**

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

```c
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.
3. Reference

**gyro**→**registerValueCallback()**

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters**:

- **callback** the callback 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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>gyro-&gt;set_highestValue()</td>
<td>Changes the recorded maximal value observed.</td>
<td>newval: a floating point number</td>
<td>YAPI_SUCCESS if the call succeeds.</td>
</tr>
<tr>
<td>int set_highestValue(double newval)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
Changes the datalogger recording frequency for this function.

```c
int set_logFrequency( string newval)
```

The frequency can be specified as samples per second, as samples 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.
Changes the logical name of the gyroscope.

```c
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.
Changes the recorded minimal value observed.

```c
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.
Changes the timed value notification frequency for this function.

```c
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.
3. Reference

Changes the resolution of the measured physical values.

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

```c
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. Reference

3.20. 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:

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

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindHubPort</code></td>
<td>Retrieves a Yocto-hub port for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstHubPort</code></td>
<td>Starts the enumeration of Yocto-hub ports currently accessible.</td>
</tr>
</tbody>
</table>

### YHubPort methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hubport-&gt;describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form <code>TYPE(NAME)=SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_advertisedValue()</code></td>
<td>Returns the current value of the Yocto-hub port (no more than 6 characters).</td>
</tr>
<tr>
<td><code>hubport-&gt;get_baudRate()</code></td>
<td>Returns the current baud rate used by this Yocto-hub port, in kbps.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_enabled()</code></td>
<td>Returns true if the Yocto-hub port is powered, false otherwise.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the Yocto-hub port.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the Yocto-hub port.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_friendlyName()</code></td>
<td>Returns a global identifier of the Yocto-hub port in the format <code>MODULE_NAME.FUNCTION_NAME</code>.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_functionId()</code></td>
<td>Returns the hardware identifier of the Yocto-hub port, without reference to the module.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the Yocto-hub port in the form <code>SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>hubport-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the Yocto-hub port.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>hubport→get_module()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>hubport→get_module_async(callback, context)</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>hubport→get_portState()</td>
<td>Returns the current state of the Yocto-hub port.</td>
</tr>
<tr>
<td>hubport→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>hubport→isOnline()</td>
<td>Checks if the Yocto-hub port is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>hubport→isOnline_async(callback, context)</td>
<td>Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>hubport→load(msValidity)</td>
<td>Preloads the Yocto-hub port cache with a specified validity duration.</td>
</tr>
<tr>
<td>hubport→load_async(msValidity, callback, context)</td>
<td>Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>hubport→nextHubPort()</td>
<td>Continues the enumeration of Yocto-hub ports started using yFirstHubPort().</td>
</tr>
<tr>
<td>hubport→registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>hubport→set_enabled(newval)</td>
<td>Changes the activation of the Yocto-hub port.</td>
</tr>
<tr>
<td>hubport→set_logicalName(newval)</td>
<td>Changes the logical name of the Yocto-hub port.</td>
</tr>
<tr>
<td>hubport→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>hubport→wait_async(callback, context)</td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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()

**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`.

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

**Returns**:

a string that describes the Yocto-hub port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
hubport → get_advertisedValue()  
hubport → advertisedValue()  
hubport.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()
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.
### YHubPort

**hubport.get_enabled()**

- **hubport.enabled()**
- **hubport.get_enabled()**

Returns true if the Yocto-hub port is powered, false otherwise.

```c
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()hubport.errorMessage()hubport.get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

string get_errorMessage()

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

Returns:
- a string corresponding to the latest error message that occurred while using the Yocto-hub port object
hubport→get_errorType()
hubport→errorType() hubport.get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

YRETCODE 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()
hubport→functionDescriptor()

hubport.get_functionDescriptor()

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

YFUN_DESCR 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.
Returns the hardware identifier of the Yocto-hub port, without reference to the module.

```cpp
string get_functionId() {
    // Returns a string that identifies the Yocto-hub port (e.g., relay1)
    // On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
}
```

For example, `relay1`
hubport->get_hardwareId()  
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()

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\rightarrow get\_module()  
 YHubPort

hubport\rightarrow module() hubport.get\_module()  

<table>
<thead>
<tr>
<th>Gets the YModule object for the device on which the function is located.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YModule get_module()</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><strong>Returns</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>an instance of YModule</td>
</tr>
</tbody>
</table>
hubport → get_portState()  
YHubPort  
hubport → portState()  
hubport.get_portState()  

Returns the current state of the Yocto-hub port.

```c
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()
hubport→userData() hubport.get_userData()

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

**object get_userData()**

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

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

<table>
<thead>
<tr>
<th>YHubPort</th>
<th>hubport→isOnline()hubport.isOnline()</th>
</tr>
</thead>
</table>

Checks if the Yocto-hub port is currently reachable, without raising any error.

```c
bool 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()

Preloads the Yocto-hub port cache with a specified validity duration.

YRET_CODE load( int 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()
hubport.registerValueCallback()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
Changes the activation of the Yocto-hub port.

```c
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.
Changes the logical name of the Yocto-hub port.

```c
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.
YHubPort
hubport→set_userData()
hubport→setUserData() hubport.set_userData()

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

```c
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. Reference

3.21. 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:

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

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindHumidity</strong>(func)</td>
</tr>
<tr>
<td>Retrieves a humidity sensor for a given identifier.</td>
</tr>
<tr>
<td><strong>yFirstHumidity</strong>()</td>
</tr>
<tr>
<td>Starts the enumeration of humidity sensors currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YHumidity methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>humidity→calibrateFromPoints</strong>(rawValues, refValues)</td>
</tr>
<tr>
<td>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.</td>
</tr>
<tr>
<td><strong>humidity→describe</strong>()</td>
</tr>
<tr>
<td>Returns a short text that describes unambiguously the instance of the humidity sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><strong>humidity→get_advertisedValue</strong>()</td>
</tr>
<tr>
<td>Returns the current value of the humidity sensor (no more than 6 characters).</td>
</tr>
<tr>
<td><strong>humidity→get_currentRawValue</strong>()</td>
</tr>
<tr>
<td>Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.</td>
</tr>
<tr>
<td><strong>humidity→get_currentValue</strong>()</td>
</tr>
<tr>
<td>Returns the current value of the humidity, in %RH, as a floating point number.</td>
</tr>
<tr>
<td><strong>humidity→get_errorMessage</strong>()</td>
</tr>
<tr>
<td>Returns the error message of the latest error with the humidity sensor.</td>
</tr>
<tr>
<td><strong>humidity→get_errorType</strong>()</td>
</tr>
<tr>
<td>Returns the numerical error code of the latest error with the humidity sensor.</td>
</tr>
<tr>
<td><strong>humidity→get_friendlyName</strong>()</td>
</tr>
<tr>
<td>Returns a global identifier of the humidity sensor in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><strong>humidity→get_functionDescriptor</strong>()</td>
</tr>
<tr>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><strong>humidity→get_functionId</strong>()</td>
</tr>
<tr>
<td>Returns the hardware identifier of the humidity sensor, without reference to the module.</td>
</tr>
<tr>
<td><strong>humidity→get_hardwareId</strong>()</td>
</tr>
<tr>
<td>Returns the unique hardware identifier of the humidity sensor in the form SERIAL.FUNCTIONID.</td>
</tr>
</tbody>
</table>
3. Reference

humidity→get_highestValue()

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

humidity→get_logFrequency()

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

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

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

humidity→get_module()

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

humidity→get_module_async(callback, context)

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

humidity→get_recordedData(startTime, endTime)

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

humidity→get_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.

humidity→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

humidity→set_logicalName(newval)

Changes the logical name of the humidity sensor.
**humidity**→**set_lowestValue**(newval)
Changes the recorded minimal value observed.

**humidity**→**set_reportFrequency**(newval)
Changes the timed value notification frequency for this function.

**humidity**→**set_resolution**(newval)
Changes the resolution of the measured physical values.

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

**humidity**→**wait_async**(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YHumidity.FindHumidity()  
yFindHumidity()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()  
\texttt{yFirstHumidity()YHumidity.FirstHumidity()}

Starts the enumeration of humidity sensors currently accessible.

\texttt{YHumidity FirstHumidity()}

Use the method \texttt{YHumidity.nextHumidity()} to iterate on next humidity sensors.

**Returns:**

- a pointer to a \texttt{YHumidity} object, corresponding to the first humidity sensor currently online, or a null pointer if there are none.
**YHumidity**

humidity→calibrateFromPoints()  

**humidity.calibrateFromPoints()**

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

```c
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
### humidity.describe()

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

<table>
<thead>
<tr>
<th>string describe( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>More precisely, <code>TYPE</code> is the type of the function, <code>NAME</code> is the name used for the first access to the function, <code>SERIAL</code> is the serial number of the module if the module is connected or &quot;unresolved&quot;, and <code>FUNCTIONID</code> is the hardware identifier of the function if the module is connected. For example, this method returns <code>Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1</code> if the module is already connected or <code>Relay(BadCustomeName.relay1)=unresolved</code> 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.</td>
</tr>
</tbody>
</table>

**Returns:**

A string that describes the humidity sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
3. Reference

humidity→get_advertisedValue()
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()
humidity→currentRawValue()
humidity.get_currentRawValue()

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

```plaintext
double get_currentRawValue()
```

**Returns:**

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

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`. 
Returns the current value of the humidity, in %RH, as a floating point number.

`double get_currentValue()`

- Returns:
  - a floating point number corresponding to the current value of the humidity, in %RH, as a floating point number

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

<table>
<thead>
<tr>
<th>humidity.get_errorMessage()</th>
<th>Returns the error message of the latest error with the humidity sensor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>humidity(errorMessage())</td>
<td>This method is mostly useful when using the Yoctopuce library with exceptions disabled.</td>
</tr>
<tr>
<td>humidity.get(errorMessage())</td>
<td>Returns: a string corresponding to the latest error message that occurred while using the humidity sensor object</td>
</tr>
</tbody>
</table>
Returns the numerical error code of the latest error with the humidity sensor.

YRETCODE 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()
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.
YHumidity

humidity→get_functionDescriptor()
 humidity→functionDescriptor()
humidity.get_functionDescriptor()

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

YFUN_DESCR 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.
YHumidity

humidity->get_functionId()

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()
humidity→highestValue()
humidity.get_highestValue()

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

double get_highestValue()

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the maximal value observed for the humidity since the device was started</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
humidity→get_logFrequency()
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.
```
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.
<table>
<thead>
<tr>
<th>humidity.get_lowestValue()</th>
<th>YHumidity humidity.get_lowestValue()</th>
</tr>
</thead>
</table>

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

```java
double get_lowestValue()
```

**Returns:**

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

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`. 
Get the `YModule` object for the device on which the function is located.

**YModule get_module()**

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

**Returns:**

an instance of `YModule`
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```java
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 measure, 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 measure, 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\rightarrow\text{get\_unit()}
humidity\rightarrow\text{unit()}
humidity\text{.get\_unit()}

<table>
<thead>
<tr>
<th><strong>YHumidity</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the measuring unit for the humidity.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String \texttt{get_unit( )}</th>
</tr>
</thead>
</table>

**Returns**:
- a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns \texttt{Y\_UNIT\_INVALID}. 
3. Reference

humidity \rightarrow \text{get\_userData()}

humidity \rightarrow \text{userData()}

humidity.get\_userData()

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

\text{object get\_userData( )}

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

\textbf{Returns :}

the object stored previously by the caller.
humidity.isOnline()  Checks if the humidity sensor is currently reachable, without raising any error.

Returns:
true if the humidity sensor can be reached, and false otherwise
Preloads the humidity sensor cache with a specified validity duration.

YHumidity

YRETCODE load( int 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msValidity</td>
<td>an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS</td>
<td>when the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
humidity→loadCalibrationPoints()
humidity.loadCalibrationPoints()

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

```plaintext
int loadCalibrationPoints(List<double> rawValues, List<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.
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()

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

```c
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()

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

```c
int registerValueCallback(ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
humidity->set_highestValue()
humidity->setHighestValue()
humidity.set_highestValue()

Changes the recorded maximal value observed.

```c
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.
humidity->set_logFrequency()
humidity->setLogFrequency()
humidity.set_logFrequency()

Changes the datalogger recording frequency for this function.

```c
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.setLogicalName()

Changes the logical name of the humidity sensor.

```c
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.
3. Reference

YHumidity

humidity → set_lowestValue()
humidity → setLowestValue()
humidity.set_lowestValue()

Changes the recorded minimal value observed.

<table>
<thead>
<tr>
<th>int set_lowestValue( double newval)</th>
</tr>
</thead>
</table>

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

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

On failure, throws an exception or returns a negative error code.
humidity->set_reportFrequency()
humidity->setReportFrequency()
humidity.set_reportFrequency()

Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

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

```c
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. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

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

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

Node.js:
```
var yoctolib = require('yoctolib');
var YLed = yoctolib.YLed;
```

PHP:
```
require_once('yoctoLed.php');
```

C++:
```
#include "yoctoLed.h"
```

Smalltalk:
```
uses yoctoLed;
```

VBScript:
```
yoctoLed.vb
```

C#:
```
yoctoLed.cs
```

Java:
```
import com.yoctopuce.YoctoAPI.YLed;
```

Python:
```
from yoctoLed 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>led.getModule_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>led.get_power()</code></td>
<td>Returns the current led state.</td>
</tr>
<tr>
<td><code>led.getUserData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>led.isOnline()</code></td>
<td>Checks if the led is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>led.isOnline_async(callback, context)</code></td>
<td>Checks if the led is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>led.load(msValidity)</code></td>
<td>Preloads the led cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>led.load_async(msValidity, callback, context)</code></td>
<td>Preloads the led cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>led.nextLed()</code></td>
<td>Continues the enumeration of leds started using <code>yFirstLed()</code>.</td>
</tr>
<tr>
<td><code>led.registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>led.set_blinking(newval)</code></td>
<td>Changes the current led signaling mode.</td>
</tr>
<tr>
<td><code>led.set_logicalName(newval)</code></td>
<td>Changes the logical name of the led.</td>
</tr>
<tr>
<td><code>led.set_luminosity(newval)</code></td>
<td>Changes the current led intensity (in per cent).</td>
</tr>
<tr>
<td><code>led.set_power(newval)</code></td>
<td>Changes the state of the led.</td>
</tr>
<tr>
<td><code>led.set_userData(data)</code></td>
<td>Stores a user context provided as argument in the <code>userData</code> attribute of the function.</td>
</tr>
<tr>
<td><code>led.wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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()**

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.
Returns a short text that describes unambiguously the instance of the led in the form

```
TYPE(NAME)=SERIAL.FUNCTIONID
```

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.

```
string describe()
```

Returns:
- A string that describes the led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
Returns the current value of the led (no more than 6 characters).

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.
3. Reference

Returns the current led signaling mode.

```c
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`. 
Returns the error message of the latest error with the led.

Returns:

- a string corresponding to the latest error message that occurred while using the led object.
Returns the numerical error code of the latest error with the led.

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

```cpp
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 example: `MyCustomNamerelay1`)

**Returns:**
- a string that uniquely identifies the led using logical names (ex: `MyCustomNamerelay1`)

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

YLed

led→get_functionDescriptor()

led→functionDescriptor()

led.get_functionDescriptor()

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

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

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()
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.

YLed
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.
3. Reference

`led->get_luminosity()`

`led->luminosity()`

`led.get_luminosity()`

Returns the current led intensity (in per cent).

```c
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()
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
Returns the current led state.

```c
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`. 
Returns the value of the userData attribute, as previously stored using method set_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.
Checks if the led is currently reachable, without raising any error.

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
Preloads the led cache with a specified validity duration.

**YRETCODE load( int 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.
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.
Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Changes the current led signaling mode.

```c
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.
Changes the logical name of the led.

```c
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.
Changes the current led intensity (in per cent).

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>newval an integer corresponding to the current led intensity (in per cent)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
led.set_power()

Changes the state of the led.

```c
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.
### 3. Reference

```c
led->set_userData()
```

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

```c
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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
yLightSensor = yoctolib.YLightSensor;
require_once('yocto_lightsensor.php');
```

```cpp
#include "yocto_lightsensor.h"
```

```m
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 uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.

#### lightsensor → get_currentValue()
- Returns the current value of the ambient light, in lux, as a floating point number.

#### 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()
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lightsensor-&gt;get_hardwareId()</strong></td>
<td>Returns the unique hardware identifier of the light sensor in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_highestValue()</strong></td>
<td>Returns the maximal value observed for the ambient light since the device was started.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_logFrequency()</strong></td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_logicalName()</strong></td>
<td>Returns the logical name of the light sensor.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_lowestValue()</strong></td>
<td>Returns the minimal value observed for the ambient light since the device was started.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_measureType()</strong></td>
<td>Returns the type of light measure.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_module()</strong></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_module_async(callback, context)</strong></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_recordedData(startTime, endTime)</strong></td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_reportFrequency()</strong></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_resolution()</strong></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_unit()</strong></td>
<td>Returns the measuring unit for the ambient light.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;get_userData()</strong></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;isOnline()</strong></td>
<td>Checks if the light sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;isOnline_async(callback, context)</strong></td>
<td>Checks if the light sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;load(msValidity)</strong></td>
<td>Preloads the light sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;loadCalibrationPoints(rawValues, refValues)</strong></td>
<td>Retrieves error correction data points previously entered using the method calibrateFromPoints.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;load_async(msValidity, callback, context)</strong></td>
<td>Preloads the light sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;nextLightSensor()</strong></td>
<td>Continues the enumeration of light sensors started using yFirstLightSensor().</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;registerTimedReportCallback(callback)</strong></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;registerValueCallback(callback)</strong></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><strong>lightsensor-&gt;set_highestValue(newval)</strong></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td>Changes the logical name of the light sensor.</td>
</tr>
<tr>
<td><code>set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>set_measureType(newval)</code></td>
<td>Modify the light sensor type used in the device.</td>
</tr>
<tr>
<td><code>set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YLightSensor.FindLightSensor()

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()
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.
YLightSensor

```cpp
lightsensor->calibrate() YLightSensor
```

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```cpp
def 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.
YLightSensor.calibrateFromPoints()

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

```c
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**

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

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
lightsensor\rightarrow\text{describe()}
lightsensor.describe()

\textbf{YLightSensor}

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

\begin{Verbatim}
string\ \text{describe()}
\end{Verbatim}

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.

\begin{Verbatim}
\textbf{Returns :}
\texttt{a string that describes the light sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)}
\end{Verbatim}
YLightSensor

Disconnect

lightsensor→get_advertisedValue()
lightsensor→advertisedValue()
lightsensor.get_advertisedValue()

<table>
<thead>
<tr>
<th>lightsensor→get_advertisedValue()</th>
<th>Returns the current value of the light sensor (no more than 6 characters).</th>
</tr>
</thead>
<tbody>
<tr>
<td>string get_advertisedValue()</td>
<td>Returns:</td>
</tr>
<tr>
<td></td>
<td>a string corresponding to the current value of the light sensor (no more than 6 characters).</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.</td>
</tr>
</tbody>
</table>
Returns the uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
YLightSensor

lightsensor→get_currentValue()
lightsensor→currentValue()
lightsensor.get_currentValue()

Returns the current value of the ambient light, in lux, as a floating point number.

double get_currentValue( )

Returns:
  a floating point number corresponding to the current value of the ambient light, in lux, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
YLightSensor
lightsensor\rightarrow get\_errorMessage()
lightsensor\rightarrow 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
<table>
<thead>
<tr>
<th>lightsensor→get_errorType()</th>
<th>YLightSensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightsensor→errorType()</td>
<td>lightsensor.get_errorType()</td>
</tr>
</tbody>
</table>

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

YRETCODE 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()
### lightsensor → friendlyName()
### lightsensor.get_friendlyName()

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

```java
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 example: `MyCustomName.relay1`)

**Returns:**
- a string that uniquely identifies the light sensor using logical names (ex: `MyCustomName.relay1`)

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

lightsensor→get_functionDescriptor()
lightsensor→functionDescriptor()
lightsensor.get_functionDescriptor()

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

YFUN_DESCR 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.
<table>
<thead>
<tr>
<th><strong>YLightSensor</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lightsensor</strong>→<strong>get_functionId()</strong></td>
</tr>
<tr>
<td><strong>lightsensor</strong>→<strong>functionId()</strong></td>
</tr>
</tbody>
</table>

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

```
string get_functionId() {
    // Returns:
    // a string that identifies the light sensor (ex: relay1)

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

For example `relay1`
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.
3. Reference

YLightSensor

lightsensor.get_highestValue()
lightsensor.highestValue()
lightsensor.get_highestValue()

Returns the maximal value observed for the ambient light since the device was started.

double get_highestValue()

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
YLightSensor

lightsensor→get_logFrequency()
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.
YLightSensor

lightsensor→get_logicalName()
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.
<table>
<thead>
<tr>
<th>lightsensor.get_lowestValue()</th>
<th>YLightSensor</th>
</tr>
</thead>
</table>

Returns the minimal value observed for the ambient light since the device was started.

```
double get_lowestValue() {
    // Returns a floating point number corresponding to the minimal value observed for the ambient light since the device was started.
    // On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
}
```
3. Reference

**YLightSensor**

```plaintext
lightsensor.get_measureType()
lightsensor.measureType()
lightsensor.get_measureType()
```

Returns the type of light measure.

```c
int get_measureType()
```

**Returns:**

A value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY` corresponding to the type of light measure.

On failure, throws an exception or returns `Y_MEASURETYPE_INVALID`. 
Gets the `YModule` object for the device on which the function is located.

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

**Returns**:
- an instance of `YModule`
lightsensor \(\rightarrow\) get_recordedData()
lightsensor \(\rightarrow\) 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 measure, 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 measure, without ending limit.

**Returns :**
- an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.
lightsensor→get_reportFrequency()
lightsensor→reportFrequency()
lightsensor.get_reportFrequency()

YLightSensor

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.
Returns the resolution of the measured values.

```c
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`. 
YLightSensor

**lightsensor.get_unit()**

*YLightSensor*

Returns the measuring unit for the ambient light.

```cpp
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**.
Returns the value of the userData attribute, as previously stored using method `set_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.
### isOnline()

YLightSensor

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

<table>
<thead>
<tr>
<th>bool isOnline( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

**Returns:**
- `true` if the light sensor can be reached, and `false` otherwise.
Preloads the light sensor cache with a specified validity duration.

YLightSensor

lightsensor\(\rightarrow\)\text{load()}\text{lightsensor}.load()

By default, whenever accessing a device, all function attributes are kept in cache for the standard 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.
YLightSensor

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

```java
int loadCalibrationPoints(List<double> rawValues, List<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.
YLightSensor nextLightSensor()

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

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.
3. Reference

YLightSensor

lightsensor.registerTimedReportCallback()

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

```c
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\rightarrow registerValueCallback() \hspace{1cm} YLightSensor

\textbf{lightsensor.registerValueCallback()}

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

\begin{verbatim}
int registerValueCallback( ValueCallback callback)
\end{verbatim}

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

\begin{tabular}{|p{10cm}|}
\hline
\textbf{Parameters} : \\
\hspace{1cm} \texttt{callback} the callback 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. \\
\hline
\end{tabular}
Reference

`lightsensor.set_highestValue()`

Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the light sensor.

```c
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.
```
YLightSensor
lightsensor.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.
Modify the light sensor type used in the device.

```c
int set_measureType( int newval)
```

The measure can either approximate the response of the human eye, focus on a specific light spectrum, depending on the capabilities of the light-sensitive cell. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**
- `newval` a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY`

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

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

**lightsensor** → **set_reportFrequency()**

Changes the timed value notification frequency for this function.

```c
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() \hspace{1cm} \text{YLightSensor}
lightsensor→setResolution() \hspace{1cm} \text{lightsensor.set_resolution()}

Changes the resolution of the measured physical values.

```c
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\rightarrow \text{set\_userData}() \hspace{1cm} \text{YLightSensor}

Lightsensor\rightarrow \text{setUserData}() 

Lightsensor.set\_userData()

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

\textbf{void set\_userData( object data)}

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

\textbf{Parameters :}

- \textit{data} any kind of object to be stored
3.24. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YMagnetometer = yoctolib.YMagnetometer;
```

```php
require_once('yocto_magnetometer.php');
```

```cpp
#include "yocto_magnetometer.h"
```

```m
uses yocto_magnetometer;
```

```vb
yocto_magnetometer.vb
```

```cs
yocto_magnetometer.cs
```

```java
import com.yoctopuce.YoctoAPI.YMagnetometer;
```

```py
from yocto_magnetometer import *
```

### Global functions

**yFindMagnetometer**(func)

Retrieves a magnetometer for a given identifier.

**yFirstMagnetometer()**

Starts the enumeration of magnetometers currently accessible.

### YMagnetometer methods

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

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

**magnetometer→describe()**

Returns a short text that describes unambiguously the instance of the magnetometer in the form

```
TYPE(NAME)=SERIAL.FUNCTIONID
```

**magnetometer→get_advertisedValue()**

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

**magnetometer→get_currentRawValue()**

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

**magnetometer→get_currentValue()**

Returns the current value of the magnetic field, in mT, as a floating point number.

**magnetometer→get_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
```
3. Reference

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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td>Changes the logical name of the magnetometer.</td>
</tr>
<tr>
<td><code>set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YMagnetometer.FindMagnetometer()

YMagnetometer.FindMagnetometer()

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.

<table>
<thead>
<tr>
<th>Parameters :</th>
</tr>
</thead>
<tbody>
<tr>
<td>func a string that uniquely characterizes the magnetometer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a YMagnetometer object allowing you to drive the magnetometer.</td>
</tr>
</tbody>
</table>
YMagnetometer.FirstMagnetometer()

YFirstMagnetometer()

YMagnetometer.FirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

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.
3. Reference

**magnetometer**→**calibrateFromPoints()**

**Y Magnetometer magnetometer.calibrateFromPoints()**

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

```cpp
int calibrateFromPoints( List<double> rawValues, List<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.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rawValues</strong> array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.</td>
</tr>
<tr>
<td><strong>refValues</strong> array of floating point numbers, corresponding to the corrected values for the correction points.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code> if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
magnetometer→describe()magnetometer.describe()

Returns a short text that describes unambiguously the instance of the magnetometer in the form

\[ \text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID} \]

\[ \text{string describe( )} \]

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

\[ \text{Returns :} \]

a string that describes the magnetometer (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
magnetometer→get_advertisedValue()
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, in mT, as a floating point number.

double get_currentRawValue()

Returns:

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
3. Reference

**magnetometer** → `get_currentValue()`

**magnetometer** → `currentValue()`

**magnetometer.get_currentValue()**

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the current value of the magnetic field, in mT, as a floating point number</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.
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()  
magnetometer → errorType()  
magnetometer.get_errorType()  

YMagnetometer

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

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

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`. 
3. Reference

YMagnetometer
magnetometer→get_functionDescriptor()
magnetometer→functionDescriptor()
magnetometer.get_functionDescriptor()

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

YFUN_DESCR 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.
YMagnetometer

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.
3. Reference

YMagnetometer
magnetometer → get_hardwareId()
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()  
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`. 
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

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.
**YMagnetometer**

**magnetometer** → **get_logicalName()**

**magnetometer** → **logicalName()**

**magnetometer.get_logicalName()**

Returns the logical name of the magnetometer.

```java
string get_logicalName()
```

**Returns**:

- a string corresponding to the logical name of the magnetometer.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
Returns the minimal value observed for the magnetic field since the device was started.

<table>
<thead>
<tr>
<th>double get_lowestValue( )</th>
</tr>
</thead>
</table>

**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()

**YModule get_module()**

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

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

**Returns:**

an instance of **YModule**
magnetometer→get_recordedData()  
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 measure, 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 measure, 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.
**YMagnetometer**

**magnetometer→get_reportFrequency()**

**magnetometer→reportFrequency()**

**magnetometer.get_reportFrequency()**

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

```java
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()**

Returns the resolution of the measured values.

```c
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`. 
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()
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\(\rightarrow\)get\_xValue()
magnetometer\(\rightarrow\)xValue()\(\rightarrow\)magnetometer.get\_xValue()

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

\[ \text{double get\_xValue( )} \]

**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.
3. Reference

**magnetometer**→**get_yValue()**

**YMagnetometer**

**magnetometer**→**yValue()**

**magnetometer.get_yValue()**

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

```java
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**.
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(magnetometer.isOnline())

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

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() \rightarrow \text{YMagnetometer}

Preloads the magnetometer cache with a specified validity duration.

\begin{Verbatim}
YRETCODE load( int msValidity)
\end{Verbatim}

By default, whenever accessing a device, all function attributes are kept in cache for the standard 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.

\begin{itemize}
  \item **Parameters** :
    \begin{itemize}
    \item \textit{msValidity} an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds
    \end{itemize}
  \item **Returns** :
    \begin{itemize}
    \item \texttt{YAPI\_SUCCESS} when the call succeeds.
    \end{itemize}
  \end{itemize}

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

**YMagnetometer**

**magnetometer.loadCalibrationPoints()**

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
### References

**YMagnetometer**

```csharp
YMagnetometer 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()  

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

```c
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()**

 YAMLagnetometer

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- **callback** the callback 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.
Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the magnetometer.

```c
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.
Changes the recorded minimal value observed.

```c
int setLowestValue( 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.
Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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.
3. Reference

magnetometer → set_userData()
magnetometer → setUserData()
magnetometer.set_userData()

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

```cpp
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
3.25. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
require_once('yocto_api.php');
```

```cpp
#include "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()
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.
Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

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()

Returns the largest value observed during the time interval covered by this measure.

```java
double get_maxValue()
```

**Returns:**

a floating-point number corresponding to the largest value observed.
YMeasure

measure \rightarrow \text{get\_minValue()}

measure \rightarrow \text{minValue()}

measure.get\_minValue()

Returns the smallest value observed during the time interval covered by this measure.

double \text{get\_minValue( )}

\textbf{Returns :}

a floating-point number corresponding to the smallest value observed.
Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

**Returns:**

- An floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.
3.26. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
require_once('yocto_api.php');
```

```cpp
#include "yocto_api.h"
```

```pas
uses yocto_api;
```

```vb
yocto_api.vb
```

```cs
yocto_api.cs
```

```java
import com.yoctopuce.YoctoAPI.YModule;
```

```py
from yocto_api import *
```

### Global functions

**yFindModule** *(func)*  
Allows you to find a module from its serial number or from its logical name.

**yFirstModule** ()  
Starts the enumeration of modules currently accessible.

### YModule methods

**module→checkFirmware**(path, onlynew)  
Test if the byn file is valid for this module.

**module→describe** ()  
Returns a descriptive text that identifies the module.

**module→download**(pathname)  
Downloads the specified built-in file and returns a binary buffer with its content.

**module→functionCount** ()  
Returns the number of functions (beside the “module” interface) available on the module.

**module→functionId**(functionIndex)  
Retrieves the hardware identifier of the n-th function on the module.

**module→functionName**(functionIndex)  
Retrieves the logical name of the n-th function on the module.

**module→functionValue**(functionIndex)  
Retrieves the advertised value of the n-th function on the module.

**module→get_allSettings** ()  
Returns all the setting of the module.

**module→get_beacon** ()  
Returns the state of the localization beacon.

**module→get_errorMessage** ()  
Returns the error message of the latest error with this module object.

**module→get_errorType** ()  
Returns the numerical error code of the latest error with this module object.

**module→get_firmwareRelease** ()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_firmwareVersion</code></td>
<td>Returns the version of the firmware embedded in the module.</td>
</tr>
<tr>
<td><code>get_hardwareId</code></td>
<td>Returns the unique hardware identifier of the module.</td>
</tr>
<tr>
<td><code>get_icon2d</code></td>
<td>Returns the icon of the module.</td>
</tr>
<tr>
<td><code>get_lastLogs</code></td>
<td>Returns a string with last logs of the module.</td>
</tr>
<tr>
<td><code>get_logicalName</code></td>
<td>Returns the logical name of the module.</td>
</tr>
<tr>
<td><code>get_luminosity</code></td>
<td>Returns the luminosity of the module informative leds (from 0 to 100).</td>
</tr>
<tr>
<td><code>get_persistentSettings</code></td>
<td>Returns the current state of persistent module settings.</td>
</tr>
<tr>
<td><code>get_productId</code></td>
<td>Returns the USB device identifier of the module.</td>
</tr>
<tr>
<td><code>get_productName</code></td>
<td>Returns the commercial name of the module, as set by the factory.</td>
</tr>
<tr>
<td><code>get_productRelease</code></td>
<td>Returns the hardware release version of the module.</td>
</tr>
<tr>
<td><code>get_rebootCountdown</code></td>
<td>Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.</td>
</tr>
<tr>
<td><code>get_serialNumber</code></td>
<td>Returns the serial number of the module, as set by the factory.</td>
</tr>
<tr>
<td><code>get_upTime</code></td>
<td>Returns the number of milliseconds spent since the module was powered on.</td>
</tr>
<tr>
<td><code>get_usbCurrent</code></td>
<td>Returns the current consumed by the module on the USB bus, in milli-amps.</td>
</tr>
<tr>
<td><code>get_userData</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>get_userVar</code></td>
<td>Returns the value previously stored in this attribute.</td>
</tr>
<tr>
<td><code>isOnline</code></td>
<td>Checks if the module is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async</code></td>
<td>Checks if the module is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the module cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the module cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextModule</code></td>
<td>Continues the module enumeration started using <code>yFirstModule()</code>.</td>
</tr>
<tr>
<td><code>reboot(secBeforeReboot)</code></td>
<td>Schedules a simple module reboot after the given number of seconds.</td>
</tr>
<tr>
<td><code>registerLogCallback</code></td>
<td>Registers a device log callback function.</td>
</tr>
</tbody>
</table>
module→revertFromFlash()
Reverts the settings stored in the nonvolatile memory, as when the module is powered on.

module→saveToFlash()
Saves the current settings in the nonvolatile memory of the module.

module→set_allSettings(settings)
Restores all the settings of the module.

module→set_beacon(newval)
Turns on or off the module localization beacon.

module→set_logicalName(newval)
Changes the logical name of the module.

module→set_luminosity(newval)
Changes the luminosity of the module informative LEDs.

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

module→set_userVar(newval)
Returns the value previously stored in this attribute.

module→triggerFirmwareUpdate(secBeforeReboot)
Schedules a module reboot into special firmware update mode.

module→updateFirmware(path)
Prepares a firmware upgrade of the module.

module→wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YModule.FindModule()

YModule.FindModule() 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()

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→checkFirmware() module.checkFirmware() YModule

Test if the byn file is valid for this module.

string checkFirmware( string path, bool onlynew)

This method is useful to test if the module need to be updated. It’s possible to pass an directory instead of a file. In this case this method return the path of the most recent appropriate byn file. If the parameter onlynew is true the function will discard firmware that are older or equal to the installed firmware.

Parameters:
- path the path of a byn file or a directory that contain byn files
- onlynew return only files that are strictly newer

Returns:
- the path of the byn file to use or a empty string if no byn files match the requirement

On failure, throws an exception or returns a string that start with "error:".
YModule

module.describe()

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
### YModule

**module->functionCount()**

Returns the number of functions (beside the "module" interface) available on the module.

```cpp
int functionCount()
```

**Returns:**
- the number of functions on the module

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

Retrieves the hardware identifier of the \( n \)th function on the module.

<table>
<thead>
<tr>
<th>module( \rightarrow )functionId() functionId</th>
</tr>
</thead>
<tbody>
<tr>
<td>string functionId(int functionIndex)</td>
</tr>
</tbody>
</table>

**Parameters:**

- `functionIndex` the index of the function for which the information is desired, starting at 0 for the first function.

**Returns:**

- a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.
Retrieves the logical name of the \( n \)th function on the module.

```c
string functionName( int functionIndex)
```

**Parameters:**
- `functionIndex` the index of the function for which the information is desired, starting at 0 for the first function.

**Returns:**
- a string corresponding to the logical name of the requested module function

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

module.functionValue()

Retrieves the advertised value of the $n$th function on the module.

string functionValue( int functionIndex)

**Parameters:**
- `functionIndex` the index of the function for which the information is desired, starting at 0 for the first function.

**Returns:**
- a short string (up to 6 characters) corresponding to the advertised value of the requested module function

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

```c
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.
```
YModule

module→get_errorMessage()

module→errorMessage()

module.get_errorMessage()

Returns the error message of the latest error with this module object.

string get_errorMessage( )

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

Returns:
a string corresponding to the latest error message that occurred while using this module object
Returns the numerical error code of the latest error with this module object.

YRETCODE 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
YModule

module→get_firmwareRelease()
module→firmwareRelease()
module.get_firmwareRelease()

This function returns the version of the firmware embedded in the module.

```cpp
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`. 
YModule

module→get_hardwareId()

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
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_lastLogs()</code></td>
<td>Returns a string with last logs of the module.</td>
</tr>
<tr>
<td><code>lastLogs()</code></td>
<td>This method returns only logs that are still in the module.</td>
</tr>
</tbody>
</table>

**Returns:**

A string with last logs of the module. On failure, throws an exception or returns `YAPI_INVALID_STRING`. 
module→get_logicalName()

Returns the logical name of the module.

Returns:
   a string corresponding to the logical name of the module

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
module\rightarrow get\_luminosity() \quad \text{YModule module\rightarrow luminosity() module.get\_luminosity()}

Returns the luminosity of the module informative leds (from 0 to 100).

\begin{verbatim}
int get_luminosity()
\end{verbatim}

**Returns:**
- an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns \texttt{Y\_LUMINOSITY\_INVALID}. 

Returns the current state of persistent module settings.

```c
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\rightarrow get\_productId() 
module\rightarrow 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.
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`. 
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 \texttt{Y\_PRODUCTRELEASE\_INVALID}. 
Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

**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`. 
Returns the serial number of the module, as set by the factory.

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.
### YModule

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>module.get_upTime()</td>
<td>Returns the number of milliseconds spent since the module was powered on.</td>
</tr>
</tbody>
</table>

#### 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`. 
Returns the current consumed by the module on the USB bus, in milli-amps.

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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>module.get_userData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>object get_userData()</code></td>
<td>This attribute is never touched directly by the API, and is at disposal of the caller to store a context.</td>
</tr>
</tbody>
</table>

**Returns:**
- the object stored previously by the caller.
module→get_userVar()

Returns the value previously stored in this attribute.

int get_userVar()

On startup and after a device reboot, the value is always reset to zero.

**Returns:**

- an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns Y_USERVAR_INVALID.
module→isOnline() → YModule

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

```cpp
bool 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()

Preloads the module cache with a specified validity duration.

YRETCODE load( int 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.
Continues the module enumeration started using `yFirstModule()`.

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() \rightarrow \text{module.reboot()}

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

```c
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.
Register a device log callback function.

```c
int registerLogCallback( LogCallback callback)
```

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

**Parameters:**
- **callback** the callback function to call, or a null pointer. The callback function should take two arguments:
  - the module object that emitted the log message,
  - the character string containing the log.
**module→revertFromFlash()**

*YModule*

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

```c
int revertFromFlash()
```

**Returns:**

- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
module\rightarrow\text{saveToFlash()}\quad \text{YModule}

Saves current settings in the nonvolatile memory of the module.

\begin{verbatim}
int saveToFlash( )
\end{verbatim}

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

\textbf{Returns :}

- \texttt{YAPI\_SUCCESS} when the call succeeds.

- On failure, throws an exception or returns a negative error code.
module→set_allSettings()
module→setAllSettings() module.set_allSettings()

Restore all the setting of the module.

int set_allSettings()

Useful to restore all the logical name and calibrations parameters of a module from a backup.

**Parameters:**
- settings a binary buffer with all settings.

**Returns:**
- YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.
Turns on or off the module localization beacon.

```c
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.
Changes the logical name of the module.

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.
Changes the luminosity of the module informative leds.

```c
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_userData()

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

```c
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
Returns the value previously stored in this attribute.

```c
int set_userVar( int newval)
```

On startup and after a device reboot, the value is always reset to zero.

**Parameters:**
- `newval` an integer

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

On failure, throws an exception or returns a negative error code.
module.triggerFirmwareUpdate() Schedules a module reboot into special firmware update mode.

Parameters:
- `secBeforeReboot` number of seconds before rebooting

Returns:
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
module → updateFirmware() → module.updateFirmware()

Prepare a firmware upgrade of the module.

YFirmwareUpdate updateFirmware( string path)

This method return a object YFirmwareUpdate which will handle the firmware upgrade process.

Parameters :
  - path the path of the byn file to use.

Returns :
  - A object YFirmwareUpdate.
3.27. Motor function interface

Yoctopuce application programming interface allows you to drive the power sent to the motor to make it turn both ways, but also to drive accelerations and decelerations. The motor will then accelerate automatically: you will not have to monitor it. The API also allows to slow down the motor by shortening its terminals: the motor will then act as an electromagnetic brake.

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

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

```nodejs
var yoctolib = require('yoctolib');
var YMotor = yoctolib.YMotor;
```

```php
require_once('yocto_motor.php');
```

```cpp
#include "yocto_motor.h"
```

```pascal
uses yocto_motor;
```

```vb
yocto_motor.vb
```

```cs
yocto_motor.cs
```

```java
import com.yoctopuce.YoctoAPI.YMotor;
```

```python
from yocto_motor import *
```

---

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindMotor</strong>&lt;br&gt;(func)</td>
</tr>
<tr>
<td><strong>yFirstMotor</strong>()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>YMotor methods</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>motor→brakingForceMove</strong>(targetPower, delay)</td>
</tr>
<tr>
<td><strong>motor→describe</strong>()</td>
</tr>
<tr>
<td><strong>motor→drivingForceMove</strong>(targetPower, delay)</td>
</tr>
<tr>
<td><strong>motor→get_advertisedValue</strong>()</td>
</tr>
<tr>
<td><strong>motor→get_brakingForce</strong>()</td>
</tr>
<tr>
<td><strong>motor→get_cutOffVoltage</strong>()</td>
</tr>
<tr>
<td><strong>motor→get_drivingForce</strong>()</td>
</tr>
<tr>
<td><strong>motor→get_errorMessage</strong>()</td>
</tr>
<tr>
<td><strong>motor→get_errorType</strong>()</td>
</tr>
<tr>
<td><strong>motor→get_failSafeTimeout</strong>()</td>
</tr>
</tbody>
</table>
motor->get_frequency()  
Returns the PWM frequency used to control the motor.

motor->get_friendlyName()  
Returns a global identifier of the motor in the format MODULE_NAME.FUNCTION_NAME.

motor->get_functionDescriptor()  
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

motor->get_hardwareId()  
Returns the hardware identifier of the motor, without reference to the module.

motor->get_hardwareId()  
Returns the unique hardware identifier of the motor in the form SERIAL,FUNCTIONID.

motor->get_logicalName()  
Returns the logical name of the motor.

motor->get_module()  
Gets the YModule object for the device on which the function is located.

motor->get_module_async(callback, context)  
Gets the YModule object for the device on which the function is located (asynchronous version).

motor->get_motorStatus()  
Returns the controller state.

motor->get_overCurrentLimit()  
Returns the current threshold (in mA) above which the controller automatically switches to error state.

motor->get_starterTime()  
Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

motor->get_userData()  
Returns the value of the userData attribute, as previously stored using method set_userData.

motor->isOnline()  
Checks if the motor is currently reachable, without raising any error.

motor->isOnline_async(callback, context)  
Checks if the motor is currently reachable, without raising any error (asynchronous version).

motor->keepALive()  
Rearms the controller failsafe timer.

motor->load(msValidity)  
Preloads the motor cache with a specified validity duration.

motor->load_async(msValidity, callback, context)  
Preloads the motor cache with a specified validity duration (asynchronous version).

motor->nextMotor()  
Continues the enumeration of motors started using yFirstMotor().

motor->registerValueCallback(callback)  
Registers the callback function that is invoked on every change of advertised value.

motor->resetStatus()  
Reset the controller state to IDLE.

motor->set_brakingForce(newval)  
Changes immediately the braking force applied to the motor (in percents).

motor->set_cutOffVoltage(newval)  
Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>motor-&gt;set_drivingForce(newval)</code></td>
<td>Changes immediately the power sent to the motor.</td>
</tr>
<tr>
<td><code>motor-&gt;set_failSafeTimeout(newval)</code></td>
<td>Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.</td>
</tr>
<tr>
<td><code>motor-&gt;set_frequency(newval)</code></td>
<td>Changes the PWM frequency used to control the motor.</td>
</tr>
<tr>
<td><code>motor-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the motor.</td>
</tr>
<tr>
<td><code>motor-&gt;set_overCurrentLimit(newval)</code></td>
<td>Changes the current threshold (in mA) above which the controller automatically switches to error state.</td>
</tr>
<tr>
<td><code>motor-&gt;set_starterTime(newval)</code></td>
<td>Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.</td>
</tr>
<tr>
<td><code>motor-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>motor-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YMotor.FindMotor()

Retrieves a motor for a given identifier.

YMotor.FindMotor(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 motor is online at the time it is invoked. The returned object is nevertheless valid. Use the method YMotor.isOnline() to test if the motor is indeed online at a given time. In case of ambiguity when looking for a motor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters:
- func a string that uniquely characterizes the motor

Returns:
- a YMotor object allowing you to drive the motor.
YMotor.FirstMotor()

Starts the enumeration of motors currently accessible.

Use the method YMotor.nextMotor() to iterate on next motors.

Returns:
a pointer to a YMotor object, corresponding to the first motor currently online, or a null pointer if there are none.
3. Reference

motor → brakingForceMove()  

Changes progressively the braking force applied to the motor for a specific duration.

```c
int brakingForceMove( double targetPower, int delay)
```

**Parameters:**
- `targetPower` desired braking force, in percents
- `delay` duration (in ms) of the transition

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

On failure, throws an exception or returns a negative error code.
motor→describe()motor.describe()

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

string describe( )

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

Returns :

a string that describes the motor (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
### motor→drivingForceMove()

Changes progressively the power sent to the moteur for a specific duration.

```c
int drivingForceMove( double targetPower, int delay)
```

**Parameters:**
- `targetPower` desired motor power, in percents (between -100% and +100%)
- `delay` duration (in ms) of the transition

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

On failure, throws an exception or returns a negative error code.
Returns the current value of the motor (no more than 6 characters).

string get_advertisedValue()

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
3. Reference

<table>
<thead>
<tr>
<th>YMotor</th>
</tr>
</thead>
<tbody>
<tr>
<td>motor→get_brakingForce()</td>
</tr>
<tr>
<td>motor→brakingForce()motor.get_brakingForce()</td>
</tr>
</tbody>
</table>

Returns the braking force applied to the motor, as a percentage.

```plaintext
double get_brakingForce()
```

The value 0 corresponds to no braking (free wheel).

**Returns:**

- a floating point number corresponding to the braking force applied to the motor, as a percentage

On failure, throws an exception or returns Y_BRAKINGFORCE_INVALID.
Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

**Returns:**
- a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

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

```java
YMotor
motor.get_cutOffVoltage()
```

This setting prevents damage to a battery that can occur when drawing current from an "empty" battery.
motor→get_drivingForce()  YMotor
motor→drivingForce()  motor.get_drivingForce()

Returns the power sent to the motor, as a percentage between -100% and +100%.

double get_drivingForce()  

Returns:
- A floating point number corresponding to the power sent to the motor, as a percentage between -100% and +100%.

On failure, throws an exception or returns Y_DRIVINGFORCE_INVALID.
motor→get_errorMessage()
motor→errorMessage()motor.get_errorMessage()

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

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 motor object
3. Reference

YMotor

```
motor->getErrorType()
```

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

```
YRETCODE get_errorType()
```

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

**Returns:**

- a number corresponding to the code of the latest error that occurred while using the motor object
YMotor

motor->get_failSafeTimeout()

motor->failSafeTimeout() motor.get_failSafeTimeout()

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

int get_failSafeTimeout()

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

**Returns:**

an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

On failure, throws an exception or returns Y_FAILSAFETIMEOUT_INVALID.
Returns the PWM frequency used to control the motor.

```cpp
double get_frequency()
```

**Returns:**

a floating point number corresponding to the PWM frequency used to control the motor

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

```cpp
motor->get_frequency()
```
motor.get_friendlyName()  

YMotor

motor.friendlyName() motor.get_friendlyName()

YMotor

<table>
<thead>
<tr>
<th>Returns</th>
<th>a string that uniquely identifies the motor using logical names (ex: MyCustomName.relay1)</th>
</tr>
</thead>
</table>

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
3. Reference

```
YMotor
motor.get_functionDescriptor()
motor.functionDescriptor()
motor.get_functionDescriptor()
```

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

YFUN_DESCR 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.
Returns the hardware identifier of the motor, without reference to the module.

string get_functionId()

For example relay1

Returns:

  a string that identifies the motor (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
Returns the unique hardware identifier of the motor in the form `SERIAL.FUNCTIONID`.

```java
string get_hardwareId()
```

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

**Returns:**
- a string that uniquely identifies the motor (ex: `RELAYLO1-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
motor->get_logicalName()
motor->logicalName() motor.get_logicalName()

Returns the logical name of the motor.

string get_logicalName( )

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

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
### 3. Reference

motor → `get_module()`

motor → `module()`

```
motor.get_module()
```

*YMotor*

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

*YModule* `get_module()`

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

**Returns:**

- an instance of *YModule*
Return the controller state.

```c
int get_motorStatus()
```

Possible states are: IDLE when the motor is stopped/in free wheel, ready to start; FORWD when the controller is driving the motor forward; BACKWD when the controller is driving the motor backward; BRAKE when the controller is braking; LOVOLT when the controller has detected a low voltage condition; HICURR when the controller has detected an overcurrent condition; HIHEAT when the controller has detected an overheat condition; FAILSF when the controller switched on the failsafe security.

When an error condition occurred (LOVOLT, HICURR, HIHEAT, FAILSF), the controller status must be explicitly reset using the `resetStatus` function.

Returns:

- a value among `Y_MOTORSTATUS_IDLE`, `Y_MOTORSTATUS_BRAKE`, `Y_MOTORSTATUS_FORWD`, `Y_MOTORSTATUS_BACKWD`, `Y_MOTORSTATUS_LOVOLT`, `Y_MOTORSTATUS_HICURR`, `Y_MOTORSTATUS_HIHEAT` and `Y_MOTORSTATUS_FAILSF`

On failure, throws an exception or returns `Y_MOTORSTATUS_INVALID`. 
3. Reference

YMotor

motor\rightarrow get\_overCurrentLimit()
motor\rightarrow overCurrentLimit()
motor.get\_overCurrentLimit()

<table>
<thead>
<tr>
<th>Returns</th>
<th>YMotor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the current threshold (in mA) above which the controller automatically switches to error state.</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{int get\_overCurrentLimit( )}
\]

A zero value means that there is no limit.

Returns:

- an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

On failure, throws an exception or returns \text{Y\_OVERCURRENTLIMIT\_INVALID}. 

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

**int get_starterTime()**

**Returns:**

an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

On failure, throws an exception or returns `Y_STARTERTIME_INVALID`. 
motor→get_userData()  
motor→userData()motor.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.
motor→isOnline()motor.isOnline()

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

bool isOnline( )

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

Returns :
true if the motor can be reached, and false otherwise
Rearms the controller failsafe timer.

```
int keepALive()
```

When the motor is running and the failsafe feature is active, this function should be called periodically to prove that the control process is running properly. Otherwise, the motor is automatically stopped after the specified timeout. Calling a motor `set` function implicitly rearms the failsafe timer.
## motor.load() motor.load()

Preloads the motor cache with a specified validity duration.

**YRETCODE load( int 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>msValidity</td>
<td>an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS</td>
<td>when the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>motor.nextMotor()</code></td>
<td>Continues the enumeration of motors started using <code>yFirstMotor()</code>. Returns a pointer to a <code>YMotor</code> object, corresponding to a motor currently online, or a null pointer if there are no more motors to enumerate.</td>
</tr>
</tbody>
</table>
motor.registerValueCallback() Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
motor\rightarrow\text{resetStatus()} \quad \text{YMotor}

Reset the controller state to IDLE.

\begin{verbatim}
int resetStatus( )
\end{verbatim}

This function must be invoked explicitly after any error condition is signaled.
Changes immediately the braking force applied to the motor (in percents).

\begin{verbatim}
int set_brakingForce( double newval)
\end{verbatim}

The value 0 corresponds to no braking (free wheel). When the braking force is changed, the driving power is set to zero. The value is a percentage.

**Parameters :**
- **newval** a floating point number corresponding to immediately the braking force applied to the motor (in percents)

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

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

motor->set_cutOffVoltage() YMotor
motor->setCutOffVoltage()motor.set_cutOffVoltage()

Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

```c
int set_cutOffVoltage( double newval)
```

This setting prevent damage to a battery that can occur when drawing current from an “empty” battery. Note that whatever the cutoff threshold, the controller switches to undervoltage error state if the power supply goes under 3V, even for a very brief time.

**Parameters :**
- `newval` a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

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

On failure, throws an exception or returns a negative error code.
motor→set_drivingForce()

Changes immediately the power sent to the motor.

**int set_drivingForce( double newval)**

The value is a percentage between -100% to 100%. If you want go easy on your mechanics and avoid excessive current consumption, try to avoid brutal power changes. For example, immediate transition from forward full power to reverse full power is a very bad idea. Each time the driving power is modified, the braking power is set to zero.

**Parameters :**

- `newval` a floating point number corresponding to immediately the power sent to the motor

**Returns :**

- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

```c
int set_failSafeTimeout( int newval)
```

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

**Parameters:**
- `newval` an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

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

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

**YMotor**

<table>
<thead>
<tr>
<th>motor→set_frequency()</th>
<th>setFrequency()</th>
<th>motor.set_frequency()</th>
</tr>
</thead>
</table>

Changes the PWM frequency used to control the motor.

```c
int set_frequency( double newval)
```

Low frequency is usually more efficient and may help the motor to start, but an audible noise might be generated. A higher frequency reduces the noise, but more energy is converted into heat.

**Parameters**:
- **newval** a floating point number corresponding to the PWM frequency used to control the motor

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

On failure, throws an exception or returns a negative error code.
Changes the logical name of the motor.

```c
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 motor.

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

On failure, throws an exception or returns a negative error code.
Changes the current threshold (in mA) above which the controller automatically switches to error state.

**YMotor**

```cpp
motor->set_overCurrentLimit()
motor->setOverCurrentLimit()
motor.set_overCurrentLimit()
```

**int set_overCurrentLimit( int newval)**

A zero value means that there is no limit. Note that whatever the current limit is, the controller switches to OVERCURRENT status if the current goes above 32A, even for a very brief time.

**Parameters :**

- **newval** an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

**Returns :**

- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

```c
int set_starterTime( int newval)
```

**Parameters:**
- `newval` an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

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

On failure, throws an exception or returns a negative error code.
Stores a user context provided as argument in the userData attribute of the function.

```c
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
3.28. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YNetwork = yoctolib.YNetwork;
```

```php
require_once('yocto_network.php');
```

```cpp
#include "yocto_network.h"
```

```m
#include "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 *
```

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindNetwork</strong>(func)</td>
</tr>
<tr>
<td>Retrieves a network interface for a given identifier.</td>
</tr>
<tr>
<td><strong>yFirstNetwork()</strong></td>
</tr>
<tr>
<td>Starts the enumeration of network interfaces currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YNetwork methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>network→callbackLogin</strong>(username, password)</td>
</tr>
<tr>
<td>Connects to the notification callback and saves the credentials required to log into it.</td>
</tr>
<tr>
<td><strong>network→describe()</strong></td>
</tr>
<tr>
<td>Returns a short text that describes unambiguously the instance of the network interface in the form \text{TYPE (NAME)=SERIAL.FUNCTIONID}.</td>
</tr>
<tr>
<td><strong>network→get_adminPassword()</strong></td>
</tr>
<tr>
<td>Returns a hash string if a password has been set for user “admin”, or an empty string otherwise.</td>
</tr>
<tr>
<td><strong>network→get_advertisedValue()</strong></td>
</tr>
<tr>
<td>Returns the current value of the network interface (no more than 6 characters).</td>
</tr>
<tr>
<td><strong>network→get_callbackCredentials()</strong></td>
</tr>
<tr>
<td>Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.</td>
</tr>
<tr>
<td><strong>network→get_callbackEncoding()</strong></td>
</tr>
<tr>
<td>Returns the encoding standard to use for representing notification values.</td>
</tr>
<tr>
<td><strong>network→get_callbackMaxDelay()</strong></td>
</tr>
<tr>
<td>Returns the maximum waiting time between two callback notifications, in seconds.</td>
</tr>
<tr>
<td><strong>network→get_callbackMethod()</strong></td>
</tr>
<tr>
<td>Returns the HTTP method used to notify callbacks for significant state changes.</td>
</tr>
<tr>
<td><strong>network→get_callbackMinDelay()</strong></td>
</tr>
<tr>
<td>Returns the minimum waiting time between two callback notifications, in seconds.</td>
</tr>
<tr>
<td><strong>network→get_callbackUrl()</strong></td>
</tr>
<tr>
<td>Returns the callback URL to notify of significant state changes.</td>
</tr>
<tr>
<td><strong>network→get_discoverable()</strong></td>
</tr>
<tr>
<td>Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td><code>network-&gt;get_errorMessage()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_errorType()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_friendlyName()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_functionDescriptor()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_functionId()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_hardwareId()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_ipAddress()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_logicalName()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_macAddress()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_module()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_module_async(callback, context)</code></td>
</tr>
<tr>
<td><code>network-&gt;get_poeCurrent()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_primaryDNS()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_readiness()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_router()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_secondaryDNS()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_subnetMask()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_userData()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_userPassword()</code></td>
</tr>
<tr>
<td><code>network-&gt;get_wwwWatchdogDelay()</code></td>
</tr>
<tr>
<td><code>network-&gt;isOnline()</code></td>
</tr>
<tr>
<td><code>network-&gt;isOnline_async(callback, context)</code></td>
</tr>
</tbody>
</table>
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()

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()
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.
Connects to the notification callback and saves the credentials required to log into it.

```c
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.
Returns a short text that describes unambiguously the instance of the network interface in the form 
\[ \text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID}. \]

```
string describe(
)
```

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

```
Returns :

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

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.
Returns the encoding standard to use for representing notification values.

```c
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`. 
### 3. Reference

<table>
<thead>
<tr>
<th>YNetwork</th>
<th>(\text{network} \rightarrow \text{get_callbackMaxDelay()})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\text{network} \rightarrow \text{callbackMaxDelay()})</td>
</tr>
<tr>
<td></td>
<td>(\text{network.get_callbackMaxDelay()})</td>
</tr>
</tbody>
</table>

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

```c
int get\_callbackMaxDelay()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Returns:</strong></td>
<td>an integer corresponding to the maximum waiting time between two callback notifications, in seconds</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns \text{Y_CALLBACKMAXDELAY_INVALID}.</td>
</tr>
</tbody>
</table>
Returns the HTTP method used to notify callbacks for significant state changes.

```cpp
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.

```c
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()
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.
Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>either <code>Y_DISCOVERABLE_FALSE</code> or  <code>Y_DISCOVERABLE_TRUE</code>, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_DISCOVERABLE_INVALID`.
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()

Returns the numerical error code of the latest error with the network interface.

Y RET CODE 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()

**Returns**

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

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

**Returns:**

A string that uniquely identifies the network interface using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
network→get_functionDescriptor()  YNetwork
network→functionDescriptor()  
network.get_functionDescriptor()  

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

YFUN_DESCR 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.*
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()

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()

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`. 
Returns the logical name of the network interface.

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string corresponding to the logical name of the network interface.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
Returns the MAC address of the network interface.

```java
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()
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 online.

Returns:
- an instance of YModule
**get_poeCurrent()**

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

```c
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()
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.


Returns the current established working mode of the network interface.

```c
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() 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\rightarrow 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()

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

\textbf{Returns :}

the object stored previously by the caller.
YNetwork

network→get_userPassword()

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

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_NET_WWWWATCHDOGDELAY_INVALID.
network→isOnline()network.isOnline()

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

```cpp
bool 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\rightarrow load() \hspace{1cm} YNetwork

Preloads the network interface cache with a specified validity duration.

YRETCODE load(int 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msValidity</td>
<td>an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS</td>
<td>when the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
Continues the enumeration of network interfaces started using `yFirstNetwork()`.

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()

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()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
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.
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.
Changes the encoding standard to use for representing notification values.

```c
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_YOCOTO_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()

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

```c
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.
Changes the HTTP method used to notify callbacks for significant state changes.

```c
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.
Changes the minimum waiting time between two callback notifications, in seconds.

```c
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.
Changes the callback URL to notify significant state changes.

```c
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.
Changes the activation state of the multicast announce protocols to allow easy discovery of the
module in the network neighborhood (uPnP/Bonjour protocol).

### 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.
Changes the logical name of the network interface.

```c
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.
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.
Changes the IP address of the secondary name server to be used by the module.

```c
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.
3. Reference

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()

Changes the password for the "user" user.

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

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.
Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

```c
int useDHCP( string fallbackIpAddr,
            int fallbackSubnetMaskLen,
            string fallbackRouter)
```

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

### Parameters:
- **fallbackIpAddr**: fallback IP address, to be used when no DHCP reply is received
- **fallbackSubnetMaskLen**: fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
- **fallbackRouter**: fallback router IP address, to be used when no DHCP reply is received

### Returns:
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the configuration of the network interface to use a static IP address.

```c
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` when the call succeeds.

On failure, throws an exception or returns a negative error code.
### 3.29. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YOsControl = yoctolib.YOsControl;
```

```php
require_once('yocto_oscontrol.php');
```

```cpp
#include "yocto_oscontrol.h"
```

```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)**
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oscontrol→get_shutdownCountdown()</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>oscontrol→get_shutdownCountdown()()</td>
<td>Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.</td>
</tr>
<tr>
<td>oscontrol→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>oscontrol→isOnline()</td>
<td>Checks if the OS control is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>oscontrol→isOnline_async(callback, context)</td>
<td>Checks if the OS control is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>oscontrol→load(msValidity)</td>
<td>Preloads the OS control cache with a specified validity duration.</td>
</tr>
<tr>
<td>oscontrol→load_async(msValidity, callback, context)</td>
<td>Preloads the OS control cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>oscontrol→nextOsControl()</td>
<td>Continues the enumeration of OS control started using yFirstOsControl().</td>
</tr>
<tr>
<td>oscontrol→registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>oscontrol→set_logicalName(newval)</td>
<td>Changes the logical name of the OS control.</td>
</tr>
<tr>
<td>oscontrol→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>oscontrol→shutdown(secBeforeShutDown)</td>
<td>Schedules an OS shutdown after a given number of seconds.</td>
</tr>
<tr>
<td>oscontrol→wait_async(callback, context)</td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
Retrieves OS control for a given identifier.

```go
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()

Starts the enumeration of OS control currently accessible.

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()  

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

Returns :  

a string that describes the OS control (ex: Relay(MyCustomName.relay1)=RELAYLO1- 
123456.relay1)
3. Reference

oscontrol→get_advertisedValue()
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.
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
Returns the numerical error code of the latest error with the OS control.

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()

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.
3. Reference

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

YFUN_DESCR 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_FUNCTION_DESCRIPTOR_INVALID.
3. Reference

YOsControl

oscontrol→get_functionId()
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.
Returns the unique hardware identifier of the OS control in the form SERIAL.FUNCTIONID.

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.
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`. 
YOsControl
oscontrol→get_module()
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 online.

Returns:
- an instance of YModule
Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

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

```cpp
bool 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
3. Reference

**oscontrol→load()**

Preloads the OS control cache with a specified validity duration.

**YRETCODE load( int 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.
Continues the enumeration of OS control started using `yFirstOsControl()`.

**Return**:  
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()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Changes the logical name of the OS control.

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

```c
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.

```c
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.
3.0. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YPower = yoctolib.YPower;
```

```php
require_once('yocto_power.php');
```

```cpp
#include "yocto_power.h"
```

```m```
```
```

```pas```
```
```

```vb
uses 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.

### $\text{YPower}$ 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 \( \text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID} \).

**power→get_advertisedValue()**

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

**power→get_cosPhi()**

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

**power→get_currentRawValue()**

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

**power→get_currentValue()**

Returns the current value of the electrical power, in Watt, as a floating point number.

**power→get_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 \( \text{MODULE}._\text{NAME}.\text{FUNCTION}._\text{NAME} \).

**power→get_functionDescriptor()**

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

**power→get_functionId()**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>power-&gt;get_hardwareId()</code></td>
<td>Returns the hardware identifier of the electrical power sensor, without reference to the module.</td>
</tr>
<tr>
<td><code>power-&gt;get_highestValue()</code></td>
<td>Returns the unique hardware identifier of the electrical power sensor in the form SERIAL_FUNCTIONID.</td>
</tr>
<tr>
<td><code>power-&gt;get_logFrequency()</code></td>
<td>Returns the maximal value observed for the electrical power since the device was started.</td>
</tr>
<tr>
<td><code>power-&gt;get_lowestValue()</code></td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the datalogger flash memory.</td>
</tr>
<tr>
<td><code>power-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the electrical power sensor.</td>
</tr>
<tr>
<td><code>power-&gt;get_meter()</code></td>
<td>Returns the minimal value observed for the electrical power since the device was started.</td>
</tr>
<tr>
<td><code>power-&gt;get_meter()</code></td>
<td>Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.</td>
</tr>
<tr>
<td><code>power-&gt;get_meterTimer()</code></td>
<td>Returns the elapsed time since last energy counter reset, in seconds.</td>
</tr>
<tr>
<td><code>power-&gt;get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>power-&gt;get_module_async(callback, context)</code></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>power-&gt;get_recordedData(startTime, endTime)</code></td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><code>power-&gt;get_reportFrequency()</code></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><code>power-&gt;get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>power-&gt;get_unit()</code></td>
<td>Returns the measuring unit for the electrical power.</td>
</tr>
<tr>
<td><code>power-&gt;get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><code>power-&gt;isOnline()</code></td>
<td>Checks if the electrical power sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>power-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>power-&gt;load(msValidity)</code></td>
<td>Preloads the electrical power sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>power-&gt;loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method calibrateFromPoints.</td>
</tr>
<tr>
<td><code>power-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>power-&gt;nextPower()</code></td>
<td>Continues the enumeration of electrical power sensors started using yFirstPower().</td>
</tr>
<tr>
<td><code>power-&gt;registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>power-&gt;registerValueCallback(callback)</code></td>
<td></td>
</tr>
</tbody>
</table>
Registers the callback function that is invoked on every change of advertised value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>power-&gt;reset()</code></td>
<td>Resets the energy counter.</td>
</tr>
<tr>
<td><code>power-&gt;set_highestValue()</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>power-&gt;set_logFrequency()</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>power-&gt;set_logicalName()</code></td>
<td>Changes the logical name of the electrical power sensor.</td>
</tr>
<tr>
<td><code>power-&gt;set_lowestValue()</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>power-&gt;set_reportFrequency()</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>power-&gt;set_resolution()</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>power-&gt;set_userData()</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>power-&gt;wait_async()</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YPower.FindPower() 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()
yFirstPower() YPower.FirstPower()

Starts the enumeration of electrical power sensors currently accessible.

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()`

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

```c
int calibrateFromPoints( List<double> rawValues,
                        List<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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rawValues</code></td>
<td>array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.</td>
</tr>
<tr>
<td><code>refValues</code></td>
<td>array of floating point numbers, corresponding to the corrected values for the correction points.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code></td>
<td>if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
power.describe()  

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

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

Returns:

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

YPower

power→get_advertisedValue()
power→advertisedValue()
power.get_advertisedValue()

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

string get_advertisedValue( )

Returns:

- a string corresponding to the current value of the electrical power sensor (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
### YPower

#### 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).

<table>
<thead>
<tr>
<th>double get_cosPhi()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns:</td>
</tr>
<tr>
<td>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)</td>
</tr>
<tr>
<td>On failure, throws an exception or returns Y_COSPHI_INVALID.</td>
</tr>
</tbody>
</table>
Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

`double get_currentRawValue()`

**Returns**

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

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

```c
power.get_currentValue()
```

Retruns the current value of the electrical power, in Watt, as a floating point number.

```c
double get_currentValue() {
    // Implementation details...
}
```

**Returns:**
- a floating point number corresponding to the current value of the electrical power, in Watt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
power.get_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
Returns the numerical error code of the latest error with the electrical power sensor.

YRET CODE 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\rightarrow \text{get\_friendlyName()}

\text{power} \rightarrow \text{friendlyName()}

\text{power.get\_friendlyName()}

\textbf{YPower}

\begin{enumerate}
\item[	ext{string get\_friendlyName() }]
\end{enumerate}

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: \texttt{MyCustomName.relay1})

\begin{enumerate}
\item[	extbf{Returns} :]
\item[a string that uniquely identifies the electrical power sensor using logical names (ex: \texttt{MyCustomName.relay1})]
\item[On failure, throws an exception or returns \texttt{Y\_FRIENDLYNAME\_INVALID}.]
\end{enumerate}
3. Reference

YPower

\begin{itemize}
\item \texttt{power→get\_functionDescriptor()}
\item \texttt{power→functionDescriptor()}
\item \texttt{power.get\_functionDescriptor()}
\end{itemize}

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

\texttt{YFUN\_DESCR get\_functionDescriptor()} \

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

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>an identifier of type \texttt{YFUN_DESCR}.</td>
</tr>
</tbody>
</table>

\texttt{If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.}
power→get_functionId()

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

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()

Returns the maximal value observed for the electrical power since the device was started.

double get_highestValue() Returns:

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
3. Reference

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.
Returns the logical name of the electrical power sensor.

```c
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`. 
Returns the minimal value observed for the electrical power since the device was started.

Returns:
- A floating point number corresponding to the minimal value observed for the electrical power since the device was started.

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`. 
Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

**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`.  

Note that this counter is reset at each start of the device.
power→get_meterTimer()
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.
**YPower**

`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 $\rightarrow$ get_recordedData()
Power $\rightarrow$ recordedData() $\rightarrow$ 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 measure, 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 measure, 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.
>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.
3. Reference

**YPower**

power→get_resolution()
power→resolution()power.get_resolution()

Returns the resolution of the measured values.

```c
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`. 
3. Reference

**YPower**

```java
power.get_unit()
power.unit()(power.get_unit())
```

Returns the measuring unit for the electrical power.

```java
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.
}
```
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()**

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

```cpp
bool 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.
YPower

Preloads the electrical power sensor cache with a specified validity duration.

YRETCODE load( int 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.
3. Reference

**YPower**

---

`power.loadCalibrationPoints()` retrieves error correction data points previously entered using the method `calibrateFromPoints()`. It can be called with the following parameters:

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
Continues the enumeration of electrical power sensors started using `yFirstPower()`.  

**YPower nextPower()**

**Returns:**
- A pointer to a `YPower` object, corresponding to an electrical power sensor currently online, or a null pointer if there are no more electrical power sensors to enumerate.
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.
3. Reference

`power.registerValueCallback()`

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

```
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
### YPower

**power->reset()**

Resets the energy counter.

```c
int reset()
```

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
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.
Changes the datalogger recording frequency for this function.

```cpp
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.
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.
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.
Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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.
Reference

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

```c
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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YPressure = yoctolib.YPressure;
```

```php
require_once('yocto_pressure.php');
```

```cpp
#include "yocto_pressure.h"
```

```m
#include "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 *
```

<table>
<thead>
<tr>
<th>Global functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindPressure(func)</code></td>
<td>Retrieves a pressure sensor for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstPressure()</code></td>
<td>Starts the enumeration of pressure sensors currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YPressure methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pressure→calibrateFromPoints(rawValues, refValues)</code></td>
<td>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.</td>
</tr>
<tr>
<td><code>pressure→describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the pressure sensor in the form <code>TYPE(NAME)=SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>pressure→get_advertisedValue()</code></td>
<td>Returns the current value of the pressure sensor (no more than 6 characters).</td>
</tr>
<tr>
<td><code>pressure→get_currentRawValue()</code></td>
<td>Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.</td>
</tr>
<tr>
<td><code>pressure→get_currentValue()</code></td>
<td>Returns the current value of the pressure, in millibar (hPa), as a floating point number.</td>
</tr>
<tr>
<td><code>pressure→get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the pressure sensor.</td>
</tr>
<tr>
<td><code>pressure→get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the pressure sensor.</td>
</tr>
<tr>
<td><code>pressure→get_friendlyName()</code></td>
<td>Returns a global identifier of the pressure sensor in the format <code>MODULE_NAME::FUNCTION_NAME</code>.</td>
</tr>
<tr>
<td><code>pressure→get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.</td>
</tr>
<tr>
<td><code>pressure→get_functionId()</code></td>
<td>Returns the hardware identifier of the pressure sensor, without reference to the module.</td>
</tr>
<tr>
<td><code>pressure→get_hardwareId()</code></td>
<td></td>
</tr>
<tr>
<td><code>pressure→get_hardwareId()</code></td>
<td></td>
</tr>
</tbody>
</table>
Returns the unique hardware identifier of the pressure sensor in the form `SERIAL_FUNCTIONID`.

`pressure→get_highestValue()`
Returns the maximal value observed for the pressure since the device was started.

`pressure→get_logFrequency()`
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

`pressure→get_logicalName()`
Returns the logical name of the pressure sensor.

`pressure→get_lowestValue()`
Returns the minimal value observed for the pressure since the device was started.

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

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

`pressure→get_recordedData(startTime, endTime)`
Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

`pressure→get_reportFrequency()`
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

`pressure→get_resolution()`
Returns the resolution of the measured values.

`pressure→get_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.

`pressure→set_logFrequency(newval)`
Changes the datalogger recording frequency for this function.

`pressure→set_logicalName(newval)`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pressure-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>pressure-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>pressure-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>pressure-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>pressure-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
3. Reference

`YPressure.FindPressure()` `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()
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.
### YPressure

**pressure → calibrateFromPoints()**

**YPressure**

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

```c
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**

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

**Returns:**

- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
pressure.describe()  

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

Returns:

a string that describes the pressure sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
3. Reference

YPressure

\[
\text{pressure} \rightarrow \text{get\_advertisedValue()}
\]

\[
\text{pressure} \rightarrow \text{advertisedValue()}
\]

\[
\text{pressure.get\_advertisedValue()}
\]

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

String \text{get\_advertisedValue()} \)

\textbf{Returns :}

- a string corresponding to the current value of the pressure sensor (no more than 6 characters).

On failure, throws an exception or returns \text{Y\_ADVERTISEDVALUE\_INVALID}.
pressure->get_currentRawValue()
pressure->currentRawValue()
power.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

```
double get_currentRawValue()
```

**Returns:**

- a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number

On failure, throws an exception or returns _Y_CURRENTRAWVALUE_INVALID_.

YPressure
pressure\rightarrow \textit{get\_currentValue()}
\textit{YPressure}

\textit{YPressure} \rightarrow \textit{currentValue()}

\textit{pressure}.\textit{get\_currentValue()}

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

double \textit{get\_currentValue()}()

\textbf{Returns :}

- a floating point number corresponding to the current value of the pressure, in millibar (hPa), as a floating point number

On failure, throws an exception or returns \texttt{Y\_CURRENTVALUE\_INVALID}. 
pressure→get_errorMessage()
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
YPressure

<table>
<thead>
<tr>
<th>pressure→get_errorType()</th>
<th>pressure→errorType()</th>
<th>pressure.get_errorType()</th>
</tr>
</thead>
</table>

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

YRETCode get_errorType()

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

**Returns:**

*a number corresponding to the code of the latest error that occurred while using the pressure sensor object*
pressure→get_friendlyName()  YPressure
pressure→friendlyName()   
pressure.get_friendlyName()  

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

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.
YPressure

pressure→get_functionDescriptor()
presure→functionDescriptor()
presure.get_functionDescriptor()

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

YFUN_DESCR 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.*
YPressure

pressure→get_functionId()
purpose→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.
YPressure

pressure→get_hardwareId()
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()
pressure→highestValue()
pressure.get_highestValue()

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

double get_highestValue()

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
pressure→get_logFrequency()
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.
YPressure

pressure \rightarrow \text{get\_logicalName()}
p\rightarrow \text{logicalName()}
p\rightarrow \text{get\_logicalName()}

Returns the logical name of the pressure sensor.

\begin{verbatim}
string \text{get\_logicalName}()
\end{verbatim}

Returns:

- a string corresponding to the logical name of the pressure sensor.

On failure, throws an exception or returns \text{Y\_LOGICALNAME\_INVALID}.
YPressure

pressure→get_lowestValue()
pressure→lowestValue()
pressure.get_lowestValue()

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

```java
double get_lowestValue()
```

**Returns:**

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

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`. 
pressure→get_module()
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 online.

Returns:

an instance of YModule
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 measure, 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 measure, 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.
**YPressure**

`pressure→get_reportFrequency()`

`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.
YPressure

pressure→get_resolution()
ppressure→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() 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.
3. Reference

**YPressure**

```python
totalPressure.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()

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

`bool 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
**YPressure**

Preloads the pressure sensor cache with a specified validity duration.

<table>
<thead>
<tr>
<th>YRETCODE</th>
<th>load( int msValidity)</th>
</tr>
</thead>
</table>

By default, whenever accessing a device, all function attributes are kept in cache for the standard 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()
pressure.loadCalibrationPoints()

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

```c
int loadCalibrationPoints( List<double> rawValues, List<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() → 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.
Register the callback function that is invoked on every periodic timed notification.

```c
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()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback`: the callback 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.
Changes the recorded maximal value observed.

```c
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.
YPressure

Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the pressure sensor.

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.
Changes the recorded minimal value observed.

```c
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.
Changes the timed value notification frequency for this function.

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

```c
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. Reference

3.32. PwmInput 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:

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

### Global functions

- **yFindPwmInput(func)**
  Retrieves a voltage sensor for a given identifier.

- **yFirstPwmInput()**
  Starts the enumeration of voltage sensors currently accessible.

### YPwmInput methods

- **pwminput→calibrateFromPoints(rawValues, refValues)**
  Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

- **pwminput→describe()**
  Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **pwminput→get_advertisedValue()**
  Returns the current value of the voltage sensor (no more than 6 characters).

- **pwminput→get_currentRawValue()**
  Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

- **pwminput→get_currentValue()**
  Returns the current value of PwmInput feature as a floating point number.

- **pwminput→get_dutyCycle()**
  Returns the PWM duty cycle, in per cents.

- **pwminput→get_errorMessage()**
  Returns the error message of the latest error with the voltage sensor.

- **pwminput→get_errorType()**
  Returns the numerical error code of the latest error with the voltage sensor.

- **pwminput→get_frequency()**
  Returns the PWM frequency in Hz.

- **pwminput→get_friendlyName()**
  Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

- **pwminput→get_functionDescriptor()**
  Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
pwminput→get_functionId()
Returns the hardware identifier of the voltage sensor, without reference to the module.

pwminput→get_hardwareId()
Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.

pwminput→get_highestValue()
Returns the maximal value observed for the voltage since the device was started.

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

pwminput→get_logicalName()
Returns the logical name of the voltage sensor.

pwminput→get_lowestValue()
Returns the minimal value observed for the voltage since the device was started.

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

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

pwminput→get_period()
Returns the PWM period in milliseconds.

pwminput→get_pulseCounter()
Returns the pulse counter value.

pwminput→get_pulseDuration()
Returns the PWM pulse length in milliseconds, as a floating point number.

pwminput→get_pulseTimer()
Returns the timer of the pulses counter (ms)

pwminput→get_pwmReportMode()
Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.

pwminput→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pwminput→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pwminput→get_resolution()
Returns the resolution of the measured values.

pwminput→get_unit()
Returns the measuring unit for the values returned by get_currentValue and callbacks.

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

pwminput→isOnline()
Checks if the voltage sensor is currently reachable, without raising any error.

pwminput→isOnline_async(callback, context)
Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

pwminput→load(millisecondsValidity)
Preloads the voltage sensor cache with a specified validity duration.

pwminput→loadCalibrationPoints(rawValues, refValues)
### 3. Reference

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

```plaintext
pwminput->load_async(msValidity, callback, context)
```

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

```plaintext
pwminput->nextPwmInput()
```

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

```plaintext
pwminput->registerTimedReportCallback(callback)
```

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

```plaintext
pwminput->registerValueCallback(callback)
```

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

```plaintext
pwminput->resetCounter()
```

Returns the pulse counter value as well as his timer.

```plaintext
pwminput->set_highestValue(newval)
```

Changes the recorded maximal value observed.

```plaintext
pwminput->set_logFrequency(newval)
```

Changes the datalogger recording frequency for this function.

```plaintext
pwminput->set_logicalName(newval)
```

Changes the logical name of the voltage sensor.

```plaintext
pwminput->set_lowestValue(newval)
```

Changes the recorded minimal value observed.

```plaintext
pwminput->set_pwmReportMode(newval)
```

Modify the parameter type (frequency/duty cycle, pulse width or edge count) returned by the `get_currentValue` function and callbacks.

```plaintext
pwminput->set_reportFrequency(newval)
```

Changes the timed value notification frequency for this function.

```plaintext
pwminput->set_resolution(newval)
```

Changes the resolution of the measured physical values.

```plaintext
pwminput->set_userData(data)
```

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

```plaintext
pwminput->wait_async(callback, context)
```

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

YPwmInput.FindPwmInput(string func)

Retrieves a voltage sensor for a given identifier.

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 YPwmInput.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 YPwmInput object allowing you to drive the voltage sensor.
3. Reference

YPwmInput.FirstPwmInput()

ypFirstPwmInput() YPwmInput.FirstPwmInput()

Starts the enumeration of voltage sensors currently accessible.

Use the method YPwmInput.nextPwmInput() to iterate on next voltage sensors.

Returns:
- a pointer to a YPwmInput object, corresponding to the first voltage sensor currently online, or a null pointer if there are none.
pwminput\(\rightarrow\)calibrateFromPoints()

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

```c
int calibrateFromPoints( List<double> rawValues,
                        List<double> refValues)
```

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

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

**Parameters :**

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

**Returns :**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
### pwminput.describe()

`pwminput.describe()`

Returns a short text that describes unambiguously the instance of the voltage sensor in the form

\[
\text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID}.
\]

#### 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`)
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.
ypwmInput→get_currentRawValue()
ypwmInput→currentRawValue()
ypwmInput.get_currentRawValue()

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

double get_currentRawValue( )

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
Returns the current value of PwmInput feature as a floating point number.

**double get_currentValue()**

Depending on the pwmReportMode setting, this can be the frequency, in Hz, the duty cycle in % or the pulse length.

**Returns**:
- a floating point number corresponding to the current value of PwmInput feature as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
## Reference

### YPwmInput

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_dutyCycle()</code></td>
<td>Returns the PWM duty cycle, in per cents.</td>
</tr>
<tr>
<td><code>dutyCycle()</code></td>
<td>Returns: a floating point number corresponding to the PWM duty cycle, in per cents. On failure, throws an exception or returns <code>Y_DUTYCYCLE_INVALID</code>.</td>
</tr>
</tbody>
</table>
** pwminput→get_errorMessage()**  
** pwminput→errorMessage()**  
** pwminput.get_errorMessage()**  

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

```cpp
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
Returns the numerical error code of the latest error with the voltage sensor.

YRETCode 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
**ypminput** → **get_frequency()**

**ypminput** → **frequency()**

**ypminput.get_frequency()**

Returns the PWM frequency in Hz.

```cpp
double get_frequency() {
    // Implementation details...
    return frequency;  // a floating point number corresponding to the PWM frequency in Hz
}
```

**Returns**: a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns **Y_FREQUENCY_INVALID**.
ypminput→get_friendlyName()
pwminput→friendlyName()
pwminput.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 example: MyCustomName.relay1)

Returns:
- a string that uniquely identifies the voltage sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
ypinput→get_functionDescriptor()
ypinput→functionDescriptor()
ypinput.get_functionDescriptor()

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

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

\[
\text{string } \text{get\_functionId}( )
\]

For example \texttt{relay1}

\textbf{Returns :}

- a string that identifies the voltage sensor (ex: \texttt{relay1})

On failure, throws an exception or returns \texttt{Y\_FUNCTIONID\_INVALID}.
Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

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`. 
Returns the maximal value observed for the voltage since the device was started.

Returns :

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

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`. 
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

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.
Returns the logical name of the voltage sensor.

```plaintext
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`. 
Returns the minimal value observed for the voltage since the device was started.

**double get_lowestValue()**

**Returns:**

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

On failure, throws an exception or returns **Y_LOWESTVALUE_INVALID**.
Gets the YModule object for the device on which the function is located.

Returns:
- an instance of YModule

If the function cannot be located on any module, the returned instance of YModule is not shown as online.
Returns the PWM period in milliseconds.

```cpp
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`. 
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_pulseCounter()</code></td>
<td>Returns the pulse counter value.</td>
</tr>
<tr>
<td><code>pulseCounter()</code></td>
<td>Actually that counter is incremented twice per period. That counter is limited to 1 billions</td>
</tr>
</tbody>
</table>

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

On failure, throws an exception or returns `Y_PULSECOUNTER_INVALID`. 
Returns the PWM pulse length in milliseconds, as a floating point number.

```
double get_pulseDuration()
```

**Returns:**

- a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`. 
### 3. Reference

<table>
<thead>
<tr>
<th>Returns:</th>
<th>an integer corresponding to the timer of the pulses counter (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>On failure, throws an exception or returns <code>Y_PULSETIMER_INVALID</code>.</td>
<td></td>
</tr>
</tbody>
</table>
3. Reference

**ypwmInput**

*pwmInput* --> *get_pwmReportMode()

*pwmInput* --> *pwmReportMode()

*pwminput* . *get_pwmReportMode()

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the *get_currentValue* function and callbacks.

```c
int get_pwmReportMode()
```

Attention

**Returns:**

A value among **Y_PWMREPORTMODE_PWM_DUTYCYCLE**, **Y_PWMREPORTMODE_PWM_FREQUENCY**, **Y_PWMREPORTMODE_PWM_PULSEDURATION** and **Y_PWMREPORTMODE_PWM_EDGECOUNT** corresponding to the parameter (frequency/duty cycle, pulse width, edges count) returned by the *get_currentValue* function and callbacks.

On failure, throws an exception or returns **Y_PWMREPORTMODE_INVALID**.
3. Reference

**YPwmInput**

- `pwinput→get_recordedData()`
- `pwinput→recordedData()`
- `pwinput.get_recordedData()`

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

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

```cpp
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`. 

Returns the resolution of the measured values.

```cpp
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.
```
Returns the measuring unit for the values returned by `get_currentValue` and callbacks.

```cpp
string get_unit()
```

That unit will change according to the `pwmReportMode` settings.

**Returns:**

- a string corresponding to the measuring unit for the values returned by `get_currentValue` and callbacks

On failure, throws an exception or returns `Y_UNIT_INVALID`. 
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.
Checks if the voltage sensor is currently reachable, without raising any error.

**Returns:**
- **true** if the voltage sensor can be reached, and **false** otherwise
ypminput.load()  

Preloads the voltage sensor cache with a specified validity duration.

YAPI_SUCCESS

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

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

Returns:
- `YAPI_SUCCESS` when the call succeeds.
### loadCalibrationPoints()

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

```c
int loadCalibrationPoints( List<double> rawValues, List<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.
Continues the enumeration of voltage sensors started using `yFirstPwmInput()`.

**Returns:**
- A pointer to a `YPwmInput` object, corresponding to a voltage sensor currently online, or a null pointer if there are no more voltage sensors to enumerate.
pwminput→registerTimedReportCallback()

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

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Returns the pulse counter value as well as his timer

```
int resetCounter() {
    // Implementation
}
```

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the recorded maximal value observed.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>newval</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
### pwminput.set_logFrequency()

Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the voltage sensor.

```c
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.
Changes the recorded minimal value observed.

**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.
Modify the parameter type (frequency/duty cycle, pulse width or edge count) returned by the
get_currentValue function and callbacks.

```c
int set_pwmReportMode( int newval)
```

The edge count value will be limited to the 6 lowest digit, for values greater than one million, use
get_pulseCounter().

**Parameters :**

- `newval` a value among
  - Y_PWMREPORTMODE_PWM_DUTYCYCLE,
  - Y_PWMREPORTMODE_PWM_FREQUENCY,
  - Y_PWMREPORTMODE_PWM_PULSEDURATION
  - Y_PWMREPORTMODE_PWM_EDGECOUNT

**Returns :**

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code></td>
<td>a floating point number corresponding to the resolution of the measured physical values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code></td>
<td>if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
Stores a user context provided as argument in the userData attribute of the function.

```cpp
void setUserData( 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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YPwmOutput = yoctolib.YPwmOutput;
```

```php
require_once('yocto_pwmoutput.php');
```

```cpp
#include "yocto_pwmoutput.h"
```

```pascal
uses yocto_pwmoutput;
```

```vb
yocto_pwmoutput.vb
```

```cs
yocto_pwmoutput.cs
```

```java
import com.yoctopuce.YoctoAPI.YPwmOutput;
```

```python
from yocto_pwmoutput import *
```

<table>
<thead>
<tr>
<th>Global functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindPwmOutput(func)</code></td>
<td>Retrieves a PWM for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstPwmOutput()</code></td>
<td>Starts the enumeration of PMWs currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YPwmOutput methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PWMoutput→describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the PWM in the form <code>TYPE(NAME)=SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>PWMoutput→dutyCycleMove(target, ms_duration)</code></td>
<td>Performs a smooth change of the pulse duration toward a given value.</td>
</tr>
<tr>
<td><code>PWMoutput→get_advertisedValue()</code></td>
<td>Returns the current value of the PWM (no more than 6 characters).</td>
</tr>
<tr>
<td><code>PWMoutput→get_dutyCycle()</code></td>
<td>Returns the PWM duty cycle, in per cents.</td>
</tr>
<tr>
<td><code>PWMoutput→get_dutyCycleAtPowerOn()</code></td>
<td>Returns the PMWs duty cycle at device power on as a floating point number between 0 and 100</td>
</tr>
<tr>
<td><code>PWMoutput→get_enabled()</code></td>
<td>Returns the state of the PMWs.</td>
</tr>
<tr>
<td><code>PWMoutput→get_enabledAtPowerOn()</code></td>
<td>Returns the state of the PWM at device power on.</td>
</tr>
<tr>
<td><code>PWMoutput→get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the PWM.</td>
</tr>
<tr>
<td><code>PWMoutput→get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the PWM.</td>
</tr>
<tr>
<td><code>PWMoutput→get_frequency()</code></td>
<td>Returns the PWM frequency in Hz.</td>
</tr>
<tr>
<td><code>PWMoutput→get_friendlyName()</code></td>
<td>Returns a global identifier of the PWM in the format <code>MODULE_NAME.FUNCTION_NAME</code>.</td>
</tr>
</tbody>
</table>
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

**pwmoutput→get_functionId()**
- Returns the hardware identifier of the PWM, without reference to the module.

**pwmoutput→get_hardwareId()**
- Returns the unique hardware identifier of the PWM in the form SERIAL_FUNCTIONID.

**pwmoutput→get_logicalName()**
- Returns the logical name of the PWM.

**pwmoutput→get_module()**
- Gets the YModule object for the device on which the function is located.

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

**pwmoutput→get_period()**
- Returns the PWM period in milliseconds.

**pwmoutput→get_pulseDuration()**
- Returns the PWM pulse length in milliseconds, as a floating point number.

**pwmoutput→get_userData()**
- Returns the value of the userData attribute, as previously stored using method set_userData.

**pwmoutput→isOnline()**
- Checks if the PWM is currently reachable, without raising any error.

**pwmoutput→isOnline_async(callback, context)**
- Checks if the PWM is currently reachable, without raising any error (asynchronous version).

**pwmoutput→load(msValidity)**
- Preloads the PWM cache with a specified validity duration.

**pwmoutput→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 transition of the pulse duration toward a given value.

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

**pwmoutput→set_dutyCycle(newval)**
- Changes the PWM duty cycle, in per cents.

**pwmoutput→set_dutyCycleAtPowerOn(newval)**
- Changes the PWM duty cycle at device power on.

**pwmoutput→set_enabled(newval)**
- Stops or starts the PWM.

**pwmoutput→set_enabledAtPowerOn(newval)**
- Changes the state of the PWM at device power on.

**pwmoutput→set_frequency(newval)**
- Changes the PWM frequency.

**pwmoutput→set_logicalName(newval)**
- Changes the logical name of the PWM.

**pwmoutput→set_period(newval)**
- Changes the PWM period in milliseconds.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwmoutput-&gt;set_pulseDuration(newval)</code></td>
<td>Changes the PWM pulse length, in milliseconds.</td>
</tr>
<tr>
<td><code>pwmoutput-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>pwmoutput-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YPwmOutput.FindPwmOutput()  
YPwmOutput.FindPwmOutput()  
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.FirstPwmOutput()**

Starts the enumeration of PWMs currently accessible.

<table>
<thead>
<tr>
<th>YPwmOutput</th>
<th>FirstPwmOutput()</th>
</tr>
</thead>
</table>

Use the method **YPwmOutput.nextPwmOutput()** to iterate on next PWMs.

**Returns:**

- A pointer to a **YPwmOutput** object, corresponding to the first PWM currently online, or a **null pointer** if there are none.
YPwmOutput

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

string describe()

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

Returns :
  a string that describes the PWM (ex: Relay(MyCustomName.relay1)=RELAYLO1-
  123456.relay1)
3. Reference

pwmoutput\rightarrow \text{dutyCycleMove()}
pwmoutput.dutyCycleMove()

Performs a smooth change of the pulse duration toward a given value.

\begin{verbatim}
int dutyCycleMove( double target, int ms_duration)
\end{verbatim}

**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.
ypwmoutput → get_advertisedValue()
pwmoutput → advertisedValue()
pwmoutput.get_advertisedValue()

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

<table>
<thead>
<tr>
<th>string get_advertisedValue()</th>
</tr>
</thead>
</table>

**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.
Returns the PWM duty cycle, in per cents.

```java
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`. 
### PWM Output

**YPwmOutput**

- `pwmoutput.get_dutyCycleAtPowerOn()`
- `pwmoutput::dutyCycleAtPowerOn()`
- `pwmoutput.get_dutyCycleAtPowerOn()`

**Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100**

```java
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`. 

Returns the state of the PWMs.

```c
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`. 
### 3. Reference

**YPwmOutput**

### PWM Output

- `pwmoutput→get_enabledAtPowerOn()`
- `pwmoutput→enabledAtPowerOn()`
- `pwmoutput.get_enabledAtPowerOn()`

Returns the state of the PWM at device power on.

<table>
<thead>
<tr>
<th>int get_enabledAtPowerOn( )</th>
</tr>
</thead>
</table>

**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`. 
Returns the error message of the latest error with the PWM.

```cpp
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
ypwmoutput → get_errorType()
pwmoutput → errorType()
pwmoutput.get_errorType()

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

YRETCODE 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
Returns the PWM frequency in Hz.

```c
double get_frequency()
```

Returns:
- a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`. 
3. Reference

**YPwmOutput**

**pwmoutput→get_friendlyName()**

**pwmoutput→friendlyName()**

**pwmoutput.get_friendlyName()**

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

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

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

**ypwmoutput**→**get_functionDescriptor()**

**ypwmoutput**→**functionDescriptor()**

**ypwmoutput.get_functionDescriptor()**

<table>
<thead>
<tr>
<th>YFUN_DESCR</th>
<th>get_functionDescriptor()</th>
</tr>
</thead>
</table>

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

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

**Retruns:**

- 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()

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

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`. 
<table>
<thead>
<tr>
<th>YPwmOutput get_logicalName()</th>
<th>PWMOutput logicalName()</th>
</tr>
</thead>
</table>

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**.
3. Reference

YPwmOutput

**pwmoutput**→**get_module()**

**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 online.

**Returns:**

an instance of YModule
Returns the PWM period in milliseconds.

```java
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`. 
3. Reference

**YPwmOutput**

`pwmoutput.get_pulseDuration()`

Returns the PWM pulse length in milliseconds, as a floating point number.

```java
double get_pulseDuration()
```

**Returns:**

- a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`. 
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.
Checks if the PWM is currently reachable, without raising any error.

**Returns:**
- `true` if the PWM can be reached, and `false` otherwise.
preloads the PWM cache with a specified validity duration.

Y RETCODE load( int 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.
Continues the enumeration of PWMs started using `ypFirstPwmOutput()`.

<table>
<thead>
<tr>
<th>YPwmOutput nextPwmOutput()</th>
</tr>
</thead>
</table>

**Returns:**

- `YPwmOutput` object, corresponding to a PWM currently online, or a null pointer if there are no more PWMs to enumerate.
YPWMOutput

Performs a smooth transition of the pulse duration toward a given value.

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Changes the PWM duty cycle, in per cents.

```c
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.
Changes the PWM duty cycle at device power on.

```c
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.
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.
3. Reference

```cpp
ypwmoutput.set_enabledAtPowerOn()
```

Changes the state of the PWM at device power on.

```cpp
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.
Changes the PWM frequency.

```c
int set_frequency( double newval)
```

The duty cycle is kept unchanged thanks to an automatic pulse width change.

**Parameters :**
- `newval` a floating point number corresponding to the PWM frequency

**Returns :**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
Changes the logical name of the PWM.

```c
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.
Changes the PWM period in milliseconds.

Parameters:
- `newval` a floating point number corresponding to the PWM period in milliseconds

Returns:
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the PWM pulse length, in milliseconds.

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

```c
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
### 3.34. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YPwmPowerSource = yoctolib.YPwmPowerSource;
```

```php
require_once('yocto_pwmpowersource.php');
```

```cpp
#include "yocto_pwmpowersource.h"
```

```m
#import "yocto_pwmpowersource.h"
```

```pas
uses yocto_pwmpowersource;
```

```vb
yocto_pwmpowersource.vb
```

```cs
yocto_pwmpowersource.cs
```

```java
import com.yoctopuce.YoctoAPI.YPwmPowerSource;
```

```py
from yocto_pwmpowersource import *
```

### Global functions

- **yFindPwmPowerSource(func)**
  Retrieves a voltage source for a given identifier.

- **yFirstPwmPowerSource()**
  Starts the enumeration of Voltage sources currently accessible.

### YPwmPowerSource methods

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

- **pwmpowersource→get_advertisedValue()**
  Returns the current value of the voltage source (no more than 6 characters).

- **pwmpowersource→get_errorMessage()**
  Returns the error message of the latest error with the voltage source.

- **pwmpowersource→get_errorType()**
  Returns the numerical error code of the latest error with the voltage source.

- **pwmpowersource→get_friendlyName()**
  Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

- **pwmpowersource→get_functionDescriptor()**
  Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **pwmpowersource→get_functionId()**
  Returns the hardware identifier of the voltage source, without reference to the module.

- **pwmpowersource→get_hardwareId()**
  Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

- **pwmpowersource→get_logicalName()**
  Returns the logical name of the voltage source.

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

- **pwmpowersource→get_module_async(callback, context)**
  Gets the `YModule` object for the device on which the function is located (asynchronous version).
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwmpowersource → get_powerMode()</code></td>
<td>Returns the selected power source for the PWM on the same device</td>
</tr>
<tr>
<td><code>pwmpowersource → get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>pwmpowersource → isOnline()</code></td>
<td>Checks if the voltage source is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>pwmpowersource → isOnline_async(callback, context)</code></td>
<td>Checks if the voltage source is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>pwmpowersource → load(msValidity)</code></td>
<td>Preloads the voltage source cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>pwmpowersource → load_async(msValidity, callback, context)</code></td>
<td>Preloads the voltage source cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>pwmpowersource → nextPwmPowerSource()</code></td>
<td>Continues the enumeration of Voltage sources started using <code>yFirstPwmPowerSource()</code>.</td>
</tr>
<tr>
<td><code>pwmpowersource → registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>pwmpowersource → set_logicalName(newval)</code></td>
<td>Changes the logical name of the voltage source.</td>
</tr>
<tr>
<td><code>pwmpowersource → set_powerMode(newval)</code></td>
<td>Changes the PWM power source.</td>
</tr>
<tr>
<td><code>pwmpowersource → set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>pwmpowersource → wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YPwmPowerSource.FindPwmPowerSource()

YPwmPowerSource.FindPwmPowerSource()

YPwmPowerSource.FindPwmPowerSource()

Retrieves a voltage source for a given identifier.

YPwmPowerSource FindPwmPowerSource(string func)

The identifier can be specified using several formats:

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

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

Parameters:
  - **func** a string that uniquely characterizes the voltage source

Returns:
  - a YPwmPowerSource object allowing you to drive the voltage source.
Starts the enumeration of Voltage sources currently accessible.

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.
YNpmPowerSource

3. Reference

YNpmPowerSource.describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form

\[ \text{TYPE}(\text{NAME}) = \text{SERIAL}.\text{FUNCTIONID}. \]

String describe()

More precisely, \( \text{TYPE} \) is the type of the function, \( \text{NAME} \) is the name used for the first access to the function, \( \text{SERIAL} \) is the serial number of the module if the module is connected or "unresolved", and \( \text{FUNCTIONID} \) is the hardware identifier of the function if the module is connected. For example, this method returns \( \text{Relay(MyCustomName.relay1)} = \text{RELAYLO1-123456.relay1} \) if the module is already connected or \( \text{Relay(BadCustomeName.relay1)} = \text{unresolved} \) 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)
Reference

YPwmPowerSource

pwmpowersource→get_advertisedValue()
pwmpowersource→advertisedValue()
pwmpowersource.get_advertisedValue()

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

<table>
<thead>
<tr>
<th>string get_advertisedValue( )</th>
</tr>
</thead>
</table>

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.
3. Reference

```cpp
YPwmPowerSource

pwmpowersource→get_errorMessage()
pwmpowersource→errorMessage()
pwmpowersource.get_errorMessage()
```

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

```cpp
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
3. Reference

**YPwmPowerSource**

`pwmpowersource.get_errorType()`

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

**YRETCode 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
3. Reference

**YPwmPowerSource**

```c
YPwmPowerSource pwmpowersource;
```

**pwmpowersource→get_friendlyName()**

```c
pwmpowersource→friendlyName();
```

**pwmpowersource.get_friendlyName()**

---

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

```c
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```.
YPwmPowerSource

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

YFUN_DESCR 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.
ypwmPowerSource->get_functionId()
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.
Returns the unique hardware identifier of the voltage source in the form SERIAL.FUNCTIONID. 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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YPWMPowerSource pwmSource = YPWMPowerSource::GetHandle();</code></td>
<td></td>
</tr>
<tr>
<td><code>pwmSource-&gt;get_logicalName()</code></td>
<td></td>
</tr>
<tr>
<td><code>pwmSource.logicalName()</code></td>
<td></td>
</tr>
<tr>
<td><code>pwmSource.get_logicalName()</code></td>
<td></td>
</tr>
</tbody>
</table>

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()

YModule get_module()

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

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

**Returns:**

an instance of YModule
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YpwmPowerSource.getPowMode()</code></td>
<td>Returns the selected power source for the PWM on the same device</td>
</tr>
<tr>
<td><code>powerMode()</code></td>
<td>Returns a value among <code>Y_POWERMODE_USB_5V</code>, <code>Y_POWERMODE_USB_3V</code>, <code>Y_POWERMODE_EXT_V</code> and <code>Y_POWERMODE_OPNDRN</code> corresponding to the selected power source for the PWM on the same device</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_POWERMODE_INVALID`. 
## Reference

```python
YPwmPowerSource

pwmpowersource.get_userData()
pwmpowersource.userData()
pwmpowersource.get_userData()
```

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

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

```cpp
bool 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
```
3. Reference

**YPwmPowerSource**

*Preloads the voltage source cache with a specified validity duration.*

**YRETCODE** `load( int 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.
### YPwmPowerSource

**nextPwmPowerSource()**

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

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
Changes the logical name of the voltage source.

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.
Changes the PWM power source.

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

```c
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
3.35. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YGyro = yoctolib.YGyro;
```

```php
require_once('yocto_gyro.php');
```

```cpp
#include "yocto_gyro.h"
```

```m
#import "yocto_gyro.h"
```

```pas
uses yocto_gyro;
```

```vb
yocto_gyro.vb
```

```cs
yocto_gyro.cs
```

```java
import com.yoctopuce.YoctoAPI.YGyro;
```

```py
from yocto_gyro import *
```

**Global functions**

- **yFindQt(func)**
  - Retrieves a quaternion component for a given identifier.

- **yFirstQt()**
  - Starts the enumeration of quaternion components currently accessible.

**YQt methods**

- **-calibrateFromPoints(rawValues, refValues)**
  - Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

- **-describe()**
  - Returns a short text that describes unambiguously the instance of the quaternion component in the form
    ```text
    TYPE(NAME)=SERIAL.FUNCTIONID
    ```

- **-get_advertisedValue()**
  - Returns the current value of the quaternion component (no more than 6 characters).

- **-get_currentRawValue()**
  - Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.

- **-get_currentValue()**
  - Returns the current value of the value, in units, as a floating point number.

- **-get_errorMessage()**
  - Returns the error message of the latest error with the quaternion component.

- **-get_errorType()**
  - Returns the numerical error code of the latest error with the quaternion component.

- **-get_friendlyName()**
  - Returns a global identifier of the quaternion component in the format
    ```text
    MODULE_NAME.FUNCTION_NAME
    ```

- **-get_functionDescriptor()**
  - Returns a unique identifier of type YFUN_DESCR corresponding to the function.

- **-get_functionId()**
  - Returns the hardware identifier of the quaternion component, without reference to the module.

- **-get_hardwareId()**
3. Reference

Returns the unique hardware identifier of the quaternion component in the form SERIAL_FUNCTIONID.

qt→get_highestValue()
Returns the maximal value observed for the value since the device was started.

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

qt→get_logicalName()
Returns the logical name of the quaternion component.

qt→get_lowestValue()
Returns the minimal value observed for the value since the device was started.

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

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

qt→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

qt→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

qt→get_resolution()
Returns the resolution of the measured values.

qt→get_unit()
Returns the measuring unit for the value.

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

qt→isOnline()
Checks if the quaternion component is currently reachable, without raising any error.

qt→isOnline_async(callback, context)
Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

qt→load(msValidity)
Preloads the quaternion component cache with a specified validity duration.

qt→loadCalibrationPoints(rawValues, refValues)
Retrieves error correction data points previously entered using the method calibrateFromPoints.

qt→load_async(msValidity, callback, context)
Preloads the quaternion component cache with a specified validity duration (asynchronous version).

qt→nextQt()
Continues the enumeration of quaternion components started using yFirstQt().

qt→registerTimedReportCallback(callback)
Registers the callback function that is invoked on every periodic timed notification.

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

qt→set_highestValue(newval)
Changes the recorded maximal value observed.

qt→set_logFrequency(newval)
Changes the datalogger recording frequency for this function.

qt→set_logicalName(newval)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_lowestValue</td>
<td>Changes the logical name of the quaternion component.</td>
</tr>
<tr>
<td>set_reportFrequency</td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td>set_resolution</td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td>set_userData</td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td>wait_async</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>wait_async</td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
3. Reference

YQt.FindQt() 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()
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()qt.calibrateFromPoints()

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

```c
int calibrateFromPoints( List<double> rawValues,
                         List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
Returns a short text that describes unambiguously the instance of the quaternion component in the form \text{TYPE (NAME)=SERIAL\_FUNCTIONID}.

More precisely, \text{TYPE} is the type of the function, \text{NAME} it the name used for the first access to the function, \text{SERIAL} is the serial number of the module if the module is connected or "unresolved", and \text{FUNCTIONID} is the hardware identifier of the function if the module is connected. For example, this method returns \text{Relay(MyCustomName\_relay1)=RELAYLO1-123456\_relay1} if the module is already connected or \text{Relay(BadCustomeName\_relay1)=unresolved} if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

\begin{itemize}
  \item \textbf{Returns :}
  \begin{itemize}
    \item a string that describes the quaternion component (ex: Relay(MyCustomName\_relay1)=RELAYLO1-123456\_relay1)
  \end{itemize}
\end{itemize}
Returns the current value of the quaternion component (no more than 6 characters).

```c++
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.
```
Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.

double get_currentRawValue()

Returns:

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
Returns the current value of the value, in units, as a floating point number.

```java
double get_currentValue()
```

**Returns:**
- a floating point number corresponding to the current value of the value, in units, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
qt → get_errorMessage()
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.
Returns the numerical error code of the latest error with the quaternion component.

YRETCODE 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()
qu.t→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.
YQt
qt→get_functionDescriptor()
qt→functionDescriptor()qt.get_functionDescriptor()

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

YFUN_DESCR get_functionDescriptor()

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

**Returns:**

- an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
 qt→get_functionId()
 qt→functionId()qt.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.
Returns the unique hardware identifier of the quaternion component in the form `SERIAL_FUNCTIONID`.

### Returns:
- A string that uniquely identifies the quaternion component (e.g., `RELAYLO1-123456.relay1`)

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

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`).
qt→get_highestValue()
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()
 qr->logFrequency() qr.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.
Returns the logical name of the quaternion component.

**Returns:**

- a string corresponding to the logical name of the quaternion component.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
Returns the minimal value observed for the value since the device was started.

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\rightarrow\text{get\_module()}
qt\rightarrow\text{module()}
qt.get\_module()

Gets the \text{YModule} object for the device on which the function is located.

\text{YModule get\_module()}

If the function cannot be located on any module, the returned instance of \text{YModule} is not shown as online.

\begin{itemize}
\item \textbf{Returns} :
\item an instance of \text{YModule}
\end{itemize}
3. Reference

YQt

qt→get_recordedData()
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 measure, 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 measure, 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.
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

\textbf{string get\_report\_Frequency( )}

\textbf{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 \texttt{Y\_REPORT\_FREQUENCY\_INVALID}. 
Returns the resolution of the measured values.

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()
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.
YQt

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()
---|---
Checks if the quaternion component is currently reachable, without raising any error.

`bool 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.
Preloads the quaternion component cache with a specified validity duration.

**YAPILoad( int 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.
YQt

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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() \(\text{YQt}\)

Continues the enumeration of quaternion components started using \(\text{yFirstQt()}\).

\[
\text{YQt nextQt( )}
\]

**Returns**: 
A pointer to a \(\text{YQt}\) object, corresponding to a quaternion component currently online, or a null pointer if there are no more quaternion components to enumerate.
qt.registerTimedReportCallback()

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

```c
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.
3. Reference

qt→registerValueCallback()
qt.registerValueCallback()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- `callback` the callback 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.
\texttt{qt\rightarrow set\_highestValue()}
\texttt{qt\rightarrow setHighestValue()qt.set\_highestValue()}

Changes the recorded maximal value observed.

\textbf{int set\_highestValue( double newval)}

\textbf{Parameters :}
- \texttt{newval} a floating point number corresponding to the recorded maximal value observed

\textbf{Returns :}
- \texttt{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
qt->set_logFrequency()
qt->setLogFrequency()qt.set_logFrequency()

Changes the datalogger recording frequency for this function.

```c
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()

Changes the logical name of the quaternion component.

```cpp
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.
Changes the recorded minimal value observed.

**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.
Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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()
qt->setUserData()qt.set_userData()

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

```c
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. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even across power cuts 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YRealTimeClock = yoctolib.YRealTimeClock;
```

```php
require_once('yocto_realtimeclock.php');
```

```cpp
#include "yocto_realtimeclock.h"
```

```m```
```
```

```pas```
uses yocto_realtimeclock;
```

```vb```
yocto_realtimeclock.vb
```

```cs```
yocto_realtimeclock.cs
```

```java```
import com.yoctopuce.YoctoAPI.YRealTimeClock;
```

```py```
from yocto_realtimeclock import *
```

### Global functions

**yFindRealTimeClock(func)**
- Retrieves a clock for a given identifier.

**yFirstRealTimeClock()**
- Starts the enumeration of clocks currently accessible.

### YRealTimeClock methods

**realtimeclock→describe()**
- Returns a short text that describes unambiguously the instance of the clock in the form **TYPE**(NAME)=SERIAL.FUNCTIONID.

**realtimeclock→get_advertisedValue()**
- Returns the current value of the clock (no more than 6 characters).

**realtimeclock→get_dateTime()**
- Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

**realtimeclock→get_errorMessage()**
- Returns the error message of the latest error with the clock.

**realtimeclock→get_errorType()**
- Returns the numerical error code of the latest error with the clock.

**realtimeclock→get_friendlyName()**
- Returns a global identifier of the clock in the format **MODULE**.**NAME**.**FUNCTION**.**NAME**.

**realtimeclock→get_functionDescriptor()**
- Returns a unique identifier of type YFUN_DESCR corresponding to the function.

**realtimeclock→get_functionId()**
- Returns the hardware identifier of the clock, without reference to the module.

**realtimeclock→get_hardwareId()**
- Returns the unique hardware identifier of the clock in the form SERIAL.FUNCTIONID.

**realtimeclock→get_logicalName()**
- Returns the logical name of the clock.

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

```plaintext
realtimeclock→get_module_async(callback, context)
```

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

```plaintext
realtimeclock→get_timeSet()
```

Returns true if the clock has been set, and false otherwise.

```plaintext
realtimeclock→get_unixTime()
```

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

```plaintext
realtimeclock→get_userData()
```

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

```plaintext
realtimeclock→get_utcOffset()
```

Returns the number of seconds between current time and UTC time (time zone).

```plaintext
realtimeclock→isOnline()
```

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

```plaintext
realtimeclock→isOnline_async(callback, context)
```

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

```plaintext
realtimeclock→load(msValidity)
```

Preloads the clock cache with a specified validity duration.

```plaintext
realtimeclock→load_async(msValidity, callback, context)
```

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

```plaintext
realtimeclock→nextRealTimeClock()
```

Continues the enumeration of clocks started using yFirstRealTimeClock().

```plaintext
realtimeclock→registerValueCallback(callback)
```

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

```plaintext
realtimeclock→set_logicalName(newval)
```

Changes the logical name of the clock.

```plaintext
realtimeclock→set_unixTime(newval)
```

Changes the current time.

```plaintext
realtimeclock→set_userData(data)
```

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

```plaintext
realtimeclock→set_utcOffset(newval)
```

Changes the number of seconds between current time and UTC time (time zone).

```plaintext
realtimeclock→wait_async(callback, context)
```

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YRealTimeClock.FindRealTimeClock()
yFindRealTimeClock()
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.FirstRealTimeClock()**

Starts the enumeration of clocks currently accessible.

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

<table>
<thead>
<tr>
<th>Returns:</th>
<th>a pointer to a <code>YRealTimeClock</code> object, corresponding to the first clock currently online, or a null pointer if there are none.</th>
</tr>
</thead>
</table>
Returns a short text that describes unambiguously the instance of the clock in the form \( \text{TYPE} (\text{NAME}) = \text{SERIAL}.\text{FUNCTIONID} \).

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

Returns:
a string that describes the clock (ex: \text{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1})
realtimeclock→get_advertisedValue()
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.
3. Reference

realtimeclock\rightarrow\text{get\_dateTime()}
realtimeclock\rightarrow\text{dateTime()}
realtimeclock.get\_dateTime()

<table>
<thead>
<tr>
<th>YRealTimeClock</th>
</tr>
</thead>
<tbody>
<tr>
<td>getting the current time in the form &quot;YYYY/MM/DD hh:mm:ss&quot;</td>
</tr>
</tbody>
</table>

**string \text{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 \text{Y\_DATETIME\_INVALID}.
realtimeclock\rightarrow get\_errorMessage()
realtimeclock\rightarrow 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
YRealTimeClock
realtimeclock→get_errorType()
realtimeclock→errorType()
realtimeclock.get_errorType()

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

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

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

YFUN_DESCR 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.*
YRealTimeClock

realtimeclock → get_functionId()
realtimeclock → functionId()
realtimeclock.get_functionId()

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

```cpp
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`. 
YRealTimeClock
realtimeclock→get_hardwareId()
realtimeclock→hardwareId()
realtimeclock.get_hardwareId()

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

```java
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`.  

Returns the logical name of the clock.

```cpp
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 \rightarrow \text{get\_module()} \quad \text{YRealTimeClock}

realtimeclock \rightarrow \text{module()} \quad \text{realtimeclock.get\_module()}

Gets the \text{YModule} object for the device on which the function is located.

\text{YModule get\_module()}()

If the function cannot be located on any module, the returned instance of \text{YModule} is not shown as online.

**Returns:**

- an instance of \text{YModule}
<table>
<thead>
<tr>
<th>realtimeclock→get_timeSet()</th>
<th>YRealTimeClock</th>
</tr>
</thead>
</table>

Returns true if the clock has been set, and false otherwise.

```c
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`. 
3. Reference

YRealTimeClock

realtimeclock->get_unixTime()
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.
Returns the value of the userData attribute, as previously stored using method `set_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.
Returns the number of seconds between current time and UTC time (time zone).

**int get_utcOffset()**

**Returns:**
- an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns Y_UTCOFFSET_INVALID.
`realtimeclock->isOnline()`

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

```cpp
bool 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
Preloads the clock cache with a specified validity duration.

YRETCODE load( int 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.
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.
YRealTimeClock

realtimeclock.registerValueCallback()

**realtimeclock.registerValueCallback()**

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- **callback** the callback 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.
Changes the logical name of the clock.

```c
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.
Changes the current time.

```c
int set_unixTime( long newval)
```

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

```c
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
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. 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:

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

```bash
var yoctolib = require('yoctolib');
var YRefFrame = yoctolib.YRefFrame;
require_once('yocto_refframe.php');
```

```bash
#include "yocto_refframe.h"
import "yocto_refframe.h"
uses yocto_refframe;
```

```bash
yocto_refframe.vb
```

```bash
yocto_refframe.cs
```

```bash
import com.yoctopuce.YoctoAPI.YRefFrame;
```

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yFindRefFrame</td>
<td>Retrieves a reference frame for a given identifier.</td>
</tr>
<tr>
<td>yFirstRefFrame</td>
<td>Starts the enumeration of reference frames currently accessible.</td>
</tr>
</tbody>
</table>

### YRefFrame methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refframe.cancel3DCalibration()</td>
<td>Aborts the sensors tridimensional calibration process et restores normal settings.</td>
</tr>
<tr>
<td>refframe.describe()</td>
<td>Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>refframe.get_3DCalibrationHint()</td>
<td>Returns instructions to proceed to the tridimensional calibration initiated with method start3DCalibration.</td>
</tr>
<tr>
<td>refframe.get_3DCalibrationLogMsg()</td>
<td>Returns the latest log message from the calibration process.</td>
</tr>
<tr>
<td>refframe.get_3DCalibrationProgress()</td>
<td>Returns the global process indicator for the tridimensional calibration initiated with method start3DCalibration.</td>
</tr>
<tr>
<td>refframe.get_3DCalibrationStage()</td>
<td>Returns index of the current stage of the calibration initiated with method start3DCalibration.</td>
</tr>
<tr>
<td>refframe.get_3DCalibrationStageProgress()</td>
<td>Returns the process indicator for the current stage of the calibration initiated with method start3DCalibration.</td>
</tr>
<tr>
<td>refframe.get_advertisedValue()</td>
<td>Returns the current value of the reference frame (no more than 6 characters).</td>
</tr>
<tr>
<td>refframe.get_bearing()</td>
<td>Returns the reference bearing used by the compass.</td>
</tr>
</tbody>
</table>
3. Reference

reframe→get_errorMessage()
Returns the error message of the latest error with the reference frame.

reframe→get_errorType()
Returns the numerical error code of the latest error with the reference frame.

reframe→get_friendlyName()
Returns a global identifier of the reference frame in the format MODULE_NAME.FUNCTION_NAME.

reframe→get_functionDescriptor()
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

reframe→get_functionId()
Returns the hardware identifier of the reference frame, without reference to the module.

reframe→get_hardwareId()
Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

reframe→get_logicalName()
Returns the logical name of the reference frame.

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

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

reframe→get_mountOrientation()
Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_mountPosition()
Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

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

reframe→isOnline()
Checks if the reference frame is currently reachable, without raising any error.

reframe→isOnline_async(callback, context)
Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reframe→load(msValidity)
Preloads the reference frame cache with a specified validity duration.

reframe→load_async(msValidity, callback, context)
Preloads the reference frame cache with a specified validity duration (asynchronous version).

reframe→more3DCalibration()
Continues the sensors tridimensional calibration process previously initiated using method start3DCalibration.

reframe→nextRefFrame()
Continues the enumeration of reference frames started using yFirstRefFrame().

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

reframe→save3DCalibration()
Applies the sensors tridimensional calibration parameters that have just been computed.

reframe→set_bearing(newval)
Changes the reference bearing used by the compass.

reframe→set_logicalName(newval)
Changes the logical name of the reference frame.

<table>
<thead>
<tr>
<th>refframe → set_mountPosition(position, orientation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes the compass and tilt sensor frame of reference.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>refframe → set_userData(data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>refframe → start3DCalibration()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiates the sensors tridimensional calibration process.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>refframe → wait_async(callback, context)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YRefFrame.FindRefFrame() YRefFrame

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.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.
Aborts the sensors tridimensional calibration process and restores normal settings.

```c
int cancel3DCalibration()
```

On failure, throws an exception or returns a negative error code.
reframe.describe() YRefFrame

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

string describe()

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

Returns:
a string that describes the reference frame (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
YRefFrame

refframe→get_3DCalibrationHint()
refframe→3DCalibrationHint()
refframe.get_3DCalibrationHint()

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

string get_3DCalibrationHint()

Returns:
a character string.
YRefFrame

```plaintext
refframe → get_3DCalibrationLogMsg()
refframe → 3DCalibrationLogMsg()
refframe.get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.
```

```plaintext
string get_3DCalibrationLogMsg()

When no new message is available, returns an empty string.
```

**Returns:**

a character string.
YRefFrame

3. Reference

refframe→get_3DCalibrationProgress()
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).
3. Reference

YRefFrame

`refframe.get_3DCalibrationStage()`

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

```c
int get_3DCalibrationStage()
```

**Returns**: an integer, growing each time a calibration stage is completed.
3. Reference

YRefFrame

YRefFrame

refframe→get_3DCalibrationStageProgress()

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).
YRefFrame

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.
### 3. Reference

**refframe→get_bearing()**

`refframe→bearing()refframe.get_bearing()`

Returns the reference bearing used by the compass.

```c
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`. 
reframe\texttt{\rightarrow get\_errorMessage()}

\texttt{reframe\rightarrow errorMessage()}

\texttt{reframe.get\_errorMessage()}

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

\texttt{string get\_errorMessage()}

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

\textbf{Returns}:

\begin{quote}
\texttt{a string corresponding to the latest error message that occurred while using the reference frame object}
\end{quote}
Returns the numerical error code of the latest error with the reference frame.

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

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()  
refframe→functionDescriptor()  
refframe.get_functionDescriptor()

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

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

For example `relay1`

**Returns:**

- a string that identifies the reference frame (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.
Returns the unique hardware identifier of the reference frame in the form \texttt{SERIAL.FUNCTIONID}.

\texttt{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 \texttt{RELAYLO1-123456.relay1}).

\textbf{Returns}:
- a string that uniquely identifies the reference frame (ex: \texttt{RELAYLO1-123456.relay1})

On failure, throws an exception or returns \texttt{Y_HARDWAREID_INVALID}. 

YRefFrame

refframe→get_logicalName()
refframe→logicalName()refframe.getLogicalName()

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.
YRefFrame

reframe.get_module()

YModule

Returns:
an instance of YModule

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

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.
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.

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.
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.
reframe→isOnline() \[ \text{isOnline()} \]

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

\[
\text{bool isOnline( )}
\]

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:**
- \text{true} if the reference frame can be reached, and \text{false} otherwise
Preloads the reference frame cache with a specified validity duration.

YRefFrame::load(int 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.
Continues the sensors tridimensional calibration process previously initiated using method \texttt{start3DCalibration}.

\begin{verbatim}
int more3DCalibration() \end{verbatim}

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method \texttt{get_3DCalibrationHint}. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.
Continues the enumeration of reference frames started using \texttt{yFirstRefFrame()}. 

Returns:

- a pointer to a \texttt{YRefFrame} object, corresponding to a reference frame currently online, or a \texttt{null} pointer if there are no more reference frames to enumerate.
reframe→registerValueCallback()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**

- **callback** the callback 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.
reframe→save3DCalibration()
reframe.save3DCalibration()

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

```c
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.
Changes the reference bearing used by the compass.

```c
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.
Changes the logical name of the reference frame.

```c
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.
3. Reference

Changes the compass and tilt sensor frame of reference.

```c
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.
reframe-\texttt{set\_userData()}

\begin{verbatim}
set_userData(object data)
\end{verbatim}

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
YRefFrame
refframe.start3DCalibration()

Initiates the sensors tridimensional calibration process.

```c
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.
3.38. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YRelay = yoctolib.YRelay;
```

```php
require_once('yocto_relay.php');
```

```cpp
#include "yocto_relay.h"
```

```m```
```pas```
```vb```
```cs```
```java```
```py```
```from yocto_relay import *```

### Global functions

<table>
<thead>
<tr>
<th>yFindRelay(func)</th>
<th>Retrieves a relay for a given identifier.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>yFirstRelay()</th>
<th>Starts the enumeration of relays currently accessible.</th>
</tr>
</thead>
</table>

### YRelay methods

<table>
<thead>
<tr>
<th>relay→delayedPulse(ms_delay, ms_duration)</th>
<th>Schedules a pulse.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→describe()</th>
<th>Returns a short text that describes unambiguously the instance of the relay in the form TYPE(NAME)=SERIAL.FUNCTIONID.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_advertisedValue()</th>
<th>Returns the current value of the relay (no more than 6 characters).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_countdown()</th>
<th>Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_errorMessage()</th>
<th>Returns the error message of the latest error with the relay.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_errorType()</th>
<th>Returns the numerical error code of the latest error with the relay.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_friendlyName()</th>
<th>Returns a global identifier of the relay in the format MODULE_NAME.FUNCTION_NAME.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_functionDescriptor()</th>
<th>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>relay→get_functionId()</th>
<th>Returns the hardware identifier of the relay, without reference to the module.</th>
</tr>
</thead>
</table>

| relay→get_hardwareId() | |
3. Reference

Returns the unique hardware identifier of the relay in the form `SERIAL_FUNCTIONID`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>relay→get_logicalName()</code></td>
<td>Returns the logical name of the relay.</td>
</tr>
<tr>
<td><code>relay→get_maxTimeOnStateA()</code></td>
<td>Returns the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.</td>
</tr>
<tr>
<td><code>relay→get_maxTimeOnStateB()</code></td>
<td>Returns the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.</td>
</tr>
<tr>
<td><code>relay→get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>relay→get_module_async(callback, context)</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>relay→get_output()</code></td>
<td>Returns the output state of the relays, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td><code>relay→get_pulseTimer()</code></td>
<td>Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.</td>
</tr>
<tr>
<td><code>relay→get_state()</code></td>
<td>Returns the state of the relays (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td><code>relay→get_stateAtPowerOn()</code></td>
<td>Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).</td>
</tr>
<tr>
<td><code>relay→get_userData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>relay→isOnline()</code></td>
<td>Checks if the relay is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>relay→isOnline_async(callback, context)</code></td>
<td>Checks if the relay is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>relay→load(msValidity)</code></td>
<td>Preloads the relay cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>relay→load_async(msValidity, callback, context)</code></td>
<td>Preloads the relay cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>relay→nextRelay()</code></td>
<td>Continues the enumeration of relays started using <code>yFirstRelay()</code>.</td>
</tr>
<tr>
<td><code>relay→pulse(ms_duration)</code></td>
<td>Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).</td>
</tr>
<tr>
<td><code>relay→registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>relay→set_logicalName(newval)</code></td>
<td>Changes the logical name of the relay.</td>
</tr>
<tr>
<td><code>relay→set_maxTimeOnStateA(newval)</code></td>
<td>Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.</td>
</tr>
<tr>
<td><code>relay→set_maxTimeOnStateB(newval)</code></td>
<td></td>
</tr>
</tbody>
</table>
Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>relay.set_output(newval)</code></td>
<td>Changes the output state of the relays, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td><code>relay.set_state(newval)</code></td>
<td>Changes the state of the relays (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td><code>relay.set_stateAtPowerOn(newval)</code></td>
<td>Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).</td>
</tr>
<tr>
<td><code>relay.set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>relay.wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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()**

**yFirstRelay()**

**YRelay.FirstRelay()**

Starts the enumeration of relays currently accessible.

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.
3. Reference

relay→delayedPulse()  relay.delayedPulse()

Schedules a pulse.

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

string describe( )

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

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

relay→get_advertisedValue()
relay→advertisedValue() relay.get_advertisedValue()

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

```java
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.
Returns the number of milliseconds remaining before a pulse (delayedPulse() call). When there is no scheduled pulse, returns zero.

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.
3. Reference

relay→get_errorMessage()
relay→errorMessage() relay.get_errorMessage()

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

string get_errorMessage( )

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

Returns:
a string corresponding to the latest error message that occurred while using the relay object
YRelay

relay → get_errorType()
relay → errorType() relay.get_errorType()

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

YRETCODE 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
**3. Reference**

**YRelay**

relay→get_friendlyName()
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 example: MyCustomName.relay1)

- **Returns**:
  - a string that uniquely identifies the relay using logical names (ex: MyCustomName.relay1)

- On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
relay→get_functionDescriptor()
relay→functionDescriptor()
relay.get_functionDescriptor()

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

YFUN_DESCR 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.
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() 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.
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`. 
Retourne the maximum time (ms) allowed for $\text{THEFUNCTIONS}$ to stay in state A before automatically switching back in to B state.

```cpp
long get_maxTimeOnStateA()
```

Zero means no maximum time.

**Returns:**

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.
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()
relay→module()relay.get_module()

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

YModule get_module()

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

Returns:

an instance of YModule
3. Reference

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

**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.
3. Reference

YRelay

relay→get_state()
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.
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.
YRelay

3. Reference

relay→get_userData()
relay→userData()relay.get_userData()

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

Object `get_userData()`

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

**Returns**:

the object stored previously by the caller.
3. Reference

YRelay

`relay.isOnline()`

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

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.
Preloads the relay cache with a specified validity duration.

YRETCODE load( int 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()`

Continues the enumeration of relays started using `yFirstRelay()`.

```cpp
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.
Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```c
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()
relay.registerValueCallback()

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

```c
int registerValueCallback( ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
3. Reference

**relay→set_logicalName()**

**YRelay**

**relay→setLogicalName()**

**relay.set_logicalName()**

Changes the logical name of the relay.

```c
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()
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.
Sets the maximum time (ms) allowed for $\text{THEFUNCTIONS}$ to stay in state B before automatically switching back in to A state.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int set_maxTimeOnStateB(long newval)</code></td>
<td><code>newval</code> an integer</td>
<td><code>YAPI_SUCCESS</code> if the call succeeds.</td>
</tr>
</tbody>
</table>

Use zero for no maximum time.
Changes the output state of the relays, when used as a simple switch (single throw).

```c
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.
Changes the state of the relays (A for the idle position, B for the active position).

```c
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.
Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

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

```c
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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
require_once('yocto_api.php');
```

```cpp
#include "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**

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, in the specified unit, as a floating point number.

**sensor→get_currentValue()**

Returns the current value of the measure, in the specified unit, as a floating point number.

**sensor→get_errorMessage()**

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

**sensor→get_errorType()**

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

**sensor→get_friendlyName()**

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

**sensor→get_functionDescriptor()**

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

**sensor→get_functionId()**

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

**sensor→get_hardwareId()**
returns the unique hardware identifier of the sensor in the form SERIAL_FUNCTIONID.

sensor→get_highestValue()
Returns the maximal value observed for the measure since the device was started.

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

sensor→get_logicalName()
Returns the logical name of the sensor.

sensor→get_lowestValue()
Returns the minimal value observed for the measure since the device was started.

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

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

sensor→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

sensor→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this
function.

sensor→get_resolution()
Returns the resolution of the measured values.

sensor→get_unit()
Returns the measuring unit for the measure.

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

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

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

sensor→load(msValidity)
Preloads the sensor cache with a specified validity duration.

sensor→loadCalibrationPoints(rawValues, refValues)
Retrieves error correction data points previously entered using the method calibrateFromPoints.

sensor→load_async(msValidity, callback, context)
Preloads the sensor cache with a specified validity duration (asynchronous version).

sensor→nextSensor()
Continues the enumeration of sensors started using yFirstSensor().

sensor→registerTimedReportCallback(callback)
Registers the callback function that is invoked on every periodic timed notification.

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

sensor→set_highestValue(newval)
Changes the recorded maximal value observed.

sensor→set_logFrequency(newval)
Changes the datalogger recording frequency for this function.

sensor→set_logicalName(newval)
Changes the logical name of the sensor.

```
sensor.set_lowestValue(newval)
```

Changes the recorded minimal value observed.

```
sensor.set_reportFrequency(newval)
```

Changes the timed value notification frequency for this function.

```
sensor.set_resolution(newval)
```

Changes the resolution of the measured physical values.

```
sensor.set_userData(data)
```

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

```
sensor.wait_async(callback, context)
```

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YSensor.FindSensor() YSensor.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.
<table>
<thead>
<tr>
<th><strong>YSensor.FirstSensor()</strong></th>
</tr>
</thead>
</table>

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()
sensor.calibrateFromPoints()

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

```java
int calibrateFromPoints(List<double> rawValues, List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
sensor.describe()  

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

**Returns:**

A string that describes the sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
sensor→get_advertisedValue()
sensor→advertisedValue()
sensor.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\rightarrow get\_currentRawValue() 

sensor\rightarrow currentRawValue() 

sensor.get_currentRawValue() 

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

\textbf{double get_currentRawValue( )}

\begin{itemize}
\item \textbf{Returns :}
\begin{itemize}
\item a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number
\end{itemize}
\end{itemize}

On failure, throws an exception or returns \texttt{Y\_CURRENTRAWVALUE\_INVALID}.
sensor → get_currentValue() &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp; &n
sensor→get_errorMessage()
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()
sensor→errorType() sensor.get_errorType()

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

YRETCODE 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()

**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 example: MyCustomName.relay1)

**Returns :**

- A string that uniquely identifies the sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
sensor→get_functionDescriptor()  
sensor→functionDescriptor()  
sensor.get_functionDescriptor()  

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

YFUN_DESCR get_functionDescriptor()  

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

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
sensor → get_functionId()
sensor → functionId() \( \rightarrow \) sensor.get_functionId()

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

```cpp
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()
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()
sensor → highestValue() sensor.get_highestValue()

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

```java
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\rightarrow & \text{get\_log\_Frequency}\() \\
\text{sensor\rightarrow & \text{log\_Frequency}\() \text{sensor\_.get\_log\_Frequency}\() \\
\hline
\text{Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.}
\end{tabular}

\text{string get\_log\_Frequency( )}

\textbf{Returns :}

\begin{itemize}
  \item a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory
\end{itemize}

\text{On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.}
Returns the logical name of the sensor.

```cpp
string get_logicalName() {
    // Returns a string corresponding to the logical name of the sensor.
    // On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
}```
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()
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 online.

Returns:

an instance of YModule
sensor→get_recordedData()
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 measure, 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 measure, 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.
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()
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.
<table>
<thead>
<tr>
<th>YSensor</th>
<th>sensor-&gt;get_unit()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sensor-&gt;unit()sensor.get_unit()</td>
</tr>
</tbody>
</table>

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.
### 3. Reference

**sensor**: `get_userData()`

**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()

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

bool 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
Preloads the sensor cache with a specified validity duration.

**YAPI_SUCCESS** when the call succeeds.

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

**YSensor**

#### `sensor.loadCalibrationPoints()`

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

```c
int loadCalibrationPoints( List<double> rawValues, List<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()  
Continues the enumeration of sensors started using yFirstSensor().

**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()

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()

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

```c
int registerValueCallback(ValueCallback callback)
```

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

**Parameters:**
- `callback` the callback 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.
Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
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()

Changes the logical name of the sensor.

```c
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.
Changes the recorded minimal value observed.

### 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()

Changes the timed value notification frequency for this function.

```c
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 (e.g. "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.
Changes the resolution of the measured physical values.

```c
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()
sensor->setUserData() sensor.set_userData()

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

```c
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. SerialPort function interface

The SerialPort function interface allows you to fully drive a Yoctopuce serial port, to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

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

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

Node.js

```javascript
var yoctolib = require('yoctolib');
var YSerialPort = yoctolib.YSerialPort;
```

```php
require_once('yocto_serialport.php');
```

```cpp
#include "yocto_serialport.h"
```

```pascal
uses yocto_serialport;
```

```vbnet
yocto_serialport.vb
```

```cs
yocto_serialport.cs
```

```java
import com.yoctopuce.YoctoAPI.YSerialPort;
```

```python
from yocto_serialport import *
```

### Global functions

- **yFindSerialPort(func)**
  
  Retrieves a serial port for a given identifier.

- **yFirstSerialPort()**
  
  Starts the enumeration of serial ports currently accessible.

### YSerialPort methods

- **serialport→describe()**
  
  Returns a short text that describes unambiguously the instance of the serial port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **serialport→get_CTS()**
  
  Read the level of the CTS line.

- **serialport→get_advertisedValue()**
  
  Returns the current value of the serial port (no more than 6 characters).

- **serialport→get_errCount()**
  
  Returns the total number of communication errors detected since last reset.

- **serialport→get_errorMessage()**
  
  Returns the error message of the latest error with the serial port.

- **serialport→get_errorType()**
  
  Returns the numerical error code of the latest error with the serial port.

- **serialport→get_friendlyName()**
  
  Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

- **serialport→get_functionDescriptor()**
  
  Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **serialport→get_functionId()**
  
  Returns the hardware identifier of the serial port, without reference to the module.

- **serialport→get_hardwareId()**
  
  Returns the unique hardware identifier of the serial port in the form `SERIAL.FUNCTIONID`.

- **serialport→get_lastMsg()**
Returns the latest message fully received (for Line, Frame and Modbus protocols).

**serialport→get_logicalName()**
Returns the logical name of the serial port.

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

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

**serialport→get_msgCount()**
Returns the total number of messages received since last reset.

**serialport→get_protocol()**
Returns the type of protocol used over the serial line, as a string.

**serialport→get_rxCount()**
Returns the total number of bytes received since last reset.

**serialport→get_serialMode()**
Returns the serial port communication parameters, as a string such as "$9600,8N1$".

**serialport→get_txCount()**
Returns the total number of bytes transmitted since last reset.

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

**serialport→isOnline()**
Checks if the serial port is currently reachable, without raising any error.

**serialport→isOnline_async(callback, context)**
Checks if the serial port is currently reachable, without raising any error (asynchronous version).

**serialport→load(msValidity)**
Preloads the serial port cache with a specified validity duration.

**serialport→load_async(msValidity, callback, context)**
Preloads the serial port cache with a specified validity duration (asynchronous version).

**serialport→modbusReadBits(slaveNo, pduAddr, nBits)**
Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

**serialport→modbusReadInputBits(slaveNo, pduAddr, nBits)**
Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

**serialport→modbusReadInputRegisters(slaveNo, pduAddr, nWords)**
Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

**serialport→modbusReadRegisters(slaveNo, pduAddr, nWords)**
Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

**serialport→modbusWriteAndReadRegisters(slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)**
Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

**serialport→modbusWriteBit(slaveNo, pduAddr, value)**
Sets a single internal bit (or coil) on a MODBUS serial device.

**serialport→modbusWriteBits(slaveNo, pduAddr, bits)**
Sets several contiguous internal bits (or coils) on a MODBUS serial device.

**serialport→modbusWriteRegister(slaveNo, pduAddr, value)**
Sets a single internal register (or holding register) on a MODBUS serial device.

**serialport→modbusWriteRegisters(slaveNo, pduAddr, values)**
3. Reference

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

serialport→nextSerialPort()
Continues the enumeration of serial ports started using yFirstSerialPort().

serialport→queryLine(query, maxWait)
Sends a text line query to the serial port, and reads the reply, if any.

serialport→queryMODBUS(slaveNo, pduBytes)
Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

serialport→readHex(nBytes)
Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

serialport→readLine()
Reads a single line (or message) from the receive buffer, starting at current stream position.

serialport→readMessages(pattern, maxWait)
Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

serialport→readStr(nChars)
Reads data from the receive buffer as a string, starting at current stream position.

serialport→read_seek(rxCountVal)
Changes the current internal stream position to the specified value.

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

serialport→reset()
Clears the serial port buffer and resets counters to zero.

serialport→set_RTS(val)
Manually sets the state of the RTS line.

serialport→set_logicalName(newval)
Changes the logical name of the serial port.

serialport→set_protocol(newval)
Changes the type of protocol used over the serial line.

serialport→set_serialMode(newval)
Changes the serial port communication parameters, with a string such as “9600,8N1”.

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

serialport→wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

serialport→writeArray(byteList)
Sends a byte sequence (provided as a list of bytes) to the serial port.

serialport→writeBin(buff)
Sends a binary buffer to the serial port, as is.

serialport→writeHex(hexString)
Sends a byte sequence (provided as a hexadecimal string) to the serial port.

serialport→writeLine(text)
Sends an ASCII string to the serial port, followed by a line break (CR LF).

serialport→writeMODBUS(hexString)
Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

serialport→writeStr(text)
Sends an ASCII string to the serial port, as is.
3. Reference

YSerialPort.FindSerialPort()

YSerialPort.FindSerialPort()

Retrieves a serial port for a given identifier.

YSerialPort.FindSerialPort(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 serial port is online at the time it is invoked. The returned object is nevertheless valid. Use the method YSerialPort.isOnline() to test if the serial port is indeed online at a given time. In case of ambiguity when looking for a serial port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters:

- **func**: a string that uniquely characterizes the serial port

Returns:

- a YSerialPort object allowing you to drive the serial port.
YSerialPort.FirstSerialPort()

yFirstSerialPort() YSerialPort.FirstSerialPort()

Starts the enumeration of serial ports currently accessible.

Use the method YSerialPort.nextSerialPort() to iterate on next serial ports.

**Returns:**

- a pointer to a YSerialPort object, corresponding to the first serial port currently online, or a null pointer if there are none.
serialport → describe() serialport.describe()

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

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

Returns:

a string that describes the serial port (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
Read the level of the CTS line.

```cpp
int get_CTS()
```

The CTS line is usually driven by the RTS signal of the connected serial device.

**Returns:**

- 1 if the CTS line is high, 0 if the CTS line is low.

On failure, throws an exception or returns a negative error code.
Returns the current value of the serial port (no more than 6 characters).

### `string get_advertisedValue()`

**Returns:**
- A string corresponding to the current value of the serial port (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
serialport\rightarrow \text{get}\_\text{errCount}()

\text{serialport}\rightarrow \text{errCount}() \text{serialport.get}\_\text{errCount}()

Returns the total number of communication errors detected since last reset.

\begin{verbatim}
int get_errCount() {
    \textbf{Returns:}
    \textbf{an integer corresponding to the total number of communication errors detected since last reset}

    \textbf{On failure, throws an exception or returns Y\_ERRCOUNT\_INVALID.}
}
\end{verbatim}
serialport→get_errorMessage()
serialport→errorMessage()
serialport.get_errorMessage()

Returns the error message of the latest error with the serial 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 serial port object
serialport → get_errorType() serialport.getErrorType()

Returns the numerical error code of the latest error with the serial port.

YRETCODE get_errorType()

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

Returns:
a number corresponding to the code of the latest error that occurred while using the serial port object
serialPort→get_friendlyName()
serialPort→friendlyName()
serialPort.get_friendlyName()

 Returns a global identifier of the serial port in the format MODULE_NAME.FUNCTION_NAME.

```cpp
string get_friendlyName()
```

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

**Returns:**

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
YSerialPort

serialport.get_functionDescriptor()

serialport→get_functionDescriptor()
serialport→functionDescriptor()
serialport.get_functionDescriptor()

---

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

`YFUN_DESCR get_functionDescriptor()`

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

**Returns:**

- an identifier of type `YFUN_DESCR`.

If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`. 
SerialPort → get_functionId()
SerialPort → functionId() SerialPort.get_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

```cpp
string get_functionId();
```

For example `relay1`

**Returns:**

- a string that identifies the serial port (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
Returns the unique hardware identifier of the serial 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 serial port (for example RELAYLO1-123456.relay1).

Returns:
- a string that uniquely identifies the serial port (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
serialport → get_lastMsg()
serialport → lastMsg() serialport.get_lastMsg()

Returns the latest message fully received (for Line, Frame and Modbus protocols).

string get_lastMsg()

Returns:
- a string corresponding to the latest message fully received (for Line, Frame and Modbus protocols)

On failure, throws an exception or returns Y_LASTMSG_INVALID.
serialport→get_logicalName()
serialport→logicalName()
serialport.get_logicalName()

Returns the logical name of the serial port.

string get_logicalName( )

Returns:
  a string corresponding to the logical name of the serial port.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
serialport → get_module() serialport → module() serialport.get_module()

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

YModule get_module()

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

Returns:
- an instance of YModule
<table>
<thead>
<tr>
<th>YSerialPort</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>serialport→get_msgCount()</td>
<td>Returns the total number of messages received since last reset.</td>
</tr>
<tr>
<td>serialport→msgCount()</td>
<td>int get_msgCount( )</td>
</tr>
<tr>
<td>serialport.get_msgCount()</td>
<td>Returns :</td>
</tr>
<tr>
<td></td>
<td>an integer corresponding to the total number of messages received since last reset</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_MSGCOUNT_INVALID.</td>
</tr>
</tbody>
</table>
serialport → get_protocol()  
serialport → protocol()  
serialport.get_protocol()

Returns the type of protocol used over the serial line, as a string.

```cpp
string get_protocol()
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

**Returns:**

- a string corresponding to the type of protocol used over the serial line, as a string

On failure, throws an exception or returns Y_PROTOCOL_INVALID.
YSerialPort

Returns the total number of bytes received since last reset.

```
int get_rxCount()
```

**Returns:**
- an integer corresponding to the total number of bytes received since last reset

On failure, throws an exception or returns Y_RXCOUNT_INVALID.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serialport.get_serialMode()</td>
<td>Returns the serial port communication parameters, as a string such as &quot;9600,8N1&quot;.</td>
</tr>
<tr>
<td>Returns :</td>
<td>a string corresponding to the serial port communication parameters, as a string such as &quot;9600,8N1&quot;</td>
</tr>
<tr>
<td>On failure, throws an exception or returns Y_SERIALMODE_INVALID.</td>
<td></td>
</tr>
</tbody>
</table>
### serialport→get_txCount()

YSerialPort

serialport→txCount() → serialport.get_txCount()

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int get_txCount()</td>
<td>Returns the total number of bytes transmitted since last reset.</td>
</tr>
</tbody>
</table>

**Returns:**

- an integer corresponding to the total number of bytes transmitted since last reset

On failure, throws an exception or returns Y_TXCOUNT_INVALID.
Returns the value of the userData attribute, as previously stored using method `set_userData`.

Returns:
the object stored previously by the caller.
YSerialPort

serialport→isOnline()serialport.isOnline()

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

bool isOnline()

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

Returns:
true if the serial port can be reached, and false otherwise
Preloads the serial port cache with a specified validity duration.

YSerialPort.load(int msValidity)

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

**Parameters :**

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

**Returns :**

- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
serialport→modbusReadBits()

serialport.modbusReadBits()

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

List<int> modbusReadBits(int slaveNo, int pduAddr, int nBits)

This method uses the MODBUS function code 0x01 (Read Coils).

**Parameters:**
- **slaveNo** the address of the slave MODBUS device to query
- **pduAddr** the relative address of the first bit/coil to read (zero-based)
- **nBits** the number of bits/coils to read

**Returns:**
- a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.
serialport → modbusReadInputBits()  
serialport.modbusReadInputBits()

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

```csharp
List<int> modbusReadInputBits(int slaveNo,
                               int pduAddr,
                               int nBits)
```

This method uses the MODBUS function code 0x02 (Read Discrete Inputs).

**Parameters:**
- `slaveNo` the address of the slave MODBUS device to query
- `pduAddr` the relative address of the first bit/input to read (zero-based)
- `nBits` the number of bits/inputs to read

**Returns:**
- a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.
serialport→modbusReadInputRegisters()          YSerialPort
serialport.modbusReadInputRegisters()

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

```csharp
List<int> modbusReadInputRegisters(int slaveNo,
                                   int pduAddr,
                                   int nWords)
```

This method uses the MODBUS function code 0x04 (Read Input Registers).

**Parameters:**
- `slaveNo`  the address of the slave MODBUS device to query
- `pduAddr` the relative address of the first input register to read (zero-based)
- `nWords`  the number of input registers to read

**Returns:**
- a vector of integers, each corresponding to one 16-bit input value.

On failure, throws an exception or returns an empty array.
YSerialPort
serialport.modbusReadRegisters()

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

```csharp
List<int> modbusReadRegisters( int slaveNo,
                                int pduAddr,
                                int nWords)
```

This method uses the MODBUS function code 0x03 (Read Holding Registers).

**Parameters:**
- `slaveNo` the address of the slave MODBUS device to query
- `pduAddr` the relative address of the first holding register to read (zero-based)
- `nWords` the number of holding registers to read

**Returns:**
- a vector of integers, each corresponding to one 16-bit register value.

On failure, throws an exception or returns an empty array.
serialport→modbusWriteAndReadRegisters()

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

```csharp
List<int> modbusWriteAndReadRegisters(int slaveNo,
                        int pduWriteAddr,
                        List<int> values,
                        int pduReadAddr,
                        int nReadWords)
```

This method uses the MODBUS function code 0x17 (Read/Write Multiple Registers).

**Parameters:**
- **slaveNo** the address of the slave MODBUS device to drive
- **pduWriteAddr** the relative address of the first internal register to set (zero-based)
- **values** the vector of 16 bit values to set
- **pduReadAddr** the relative address of the first internal register to read (zero-based)
- **nReadWords** the number of 16 bit values to read

**Returns:**
a vector of integers, each corresponding to one 16-bit register value read.

On failure, throws an exception or returns an empty array.
3. Reference

`serialport → modbusWriteBit()`

Sets a single internal bit (or coil) on a MODBUS serial device.

```c
int modbusWriteBit(int slaveNo, int pduAddr, int value)
```

This method uses the MODBUS function code 0x05 (Write Single Coil).

**Parameters**:
- `slaveNo` the address of the slave MODBUS device to drive
- `pduAddr` the relative address of the bit/coil to set (zero-based)
- `value` the value to set (0 for OFF state, non-zero for ON state)

**Returns**:
- the number of bits/coils affected on the device (1)

On failure, throws an exception or returns zero.
Sets several contiguous internal bits (or coils) on a MODBUS serial device.

```csharp
int modbusWriteBits( int slaveNo, int pduAddr, List<int> bits)
```

This method uses the MODBUS function code 0x0f (Write Multiple Coils).

**Parameters:**
- `slaveNo` the address of the slave MODBUS device to drive
- `pduAddr` the relative address of the first bit/coil to set (zero-based)
- `bits` the vector of bits to be set (one integer per bit)

**Returns:**
the number of bits/coils affected on the device

On failure, throws an exception or returns zero.
Sets a single internal register (or holding register) on a MODBUS serial device.

\[
\text{modbusWriteRegister}( \text{slaveNo}, \text{pduAddr}, \text{value})
\]

This method uses the MODBUS function code 0x06 (Write Single Register).

**Parameters:**
- `slaveNo` the address of the slave MODBUS device to drive
- `pduAddr` the relative address of the register to set (zero-based)
- `value` the 16 bit value to set

**Returns:**
- the number of registers affected on the device (1)

On failure, throws an exception or returns zero.
serialport → modbusWriteRegisters()  
serialport.modbusWriteRegisters()

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

```c
int modbusWriteRegisters( int slaveNo,
                          int pduAddr,
                          List<int> values)
```

This method uses the MODBUS function code 0x10 (Write Multiple Registers).

**Parameters:**

- `slaveNo` the address of the slave MODBUS device to drive
- `pduAddr` the relative address of the first internal register to set (zero-based)
- `values` the vector of 16 bit values to set

**Returns:**

the number of registers affected on the device

On failure, throws an exception or returns zero.
serialPort -&gt; nextSerialPort() | serialPort.nextSerialPort() | YSerialPort

Continues the enumeration of serial ports started using yFirstSerialPort().

YSerialPort nextSerialPort()

Returns:

- a pointer to a YSerialPort object, corresponding to a serial port currently online, or a null pointer if there are no more serial ports to enumerate.
sends a text line query to the serial port, and reads the reply, if any.

```java
string queryLine(string query, int maxWait)
```

This function can only be used when the serial port is configured for 'Line' protocol.

**Parameters:**
- `query`    the line query to send (without CR/LF)
- `maxWait`  the maximum number of milliseconds to wait for a reply.

**Returns:**
- the next text line received after sending the text query, as a string. Additional lines can be obtained by calling `readLine` or `readMessages`.

On failure, throws an exception or returns an empty array.
YSerialPort

```
serialport.queryMODBUS()
```

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

```
List<int> queryMODBUS(int slaveNo, List<int> pduBytes)
```

The message is the PDU, provided as a vector of bytes.

**Parameters:**
- `slaveNo` the address of the slave MODBUS device to query
- `pduBytes` the message to send (PDU), as a vector of bytes. The first byte of the PDU is the MODBUS function code.

**Returns:**
- the received reply, as a vector of bytes.

On failure, throws an exception or returns an empty array (or a MODBUS error reply).
Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

```
string readHex(int nBytes)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

**Parameters:**
- `nBytes` the maximum number of bytes to read

**Returns:**
- a string with receive buffer contents, encoded in hexadecimal

On failure, throws an exception or returns a negative error code.
serialport→readLine() serialport.readLine() YSerialPort

Reads a single line (or message) from the receive buffer, starting at current stream position.

string readline()

This function can only be used when the serial port is configured for a message protocol, such as 'Line' mode or MODBUS protocols. It does not work in plain stream modes, eg. 'Char' or 'Byte').

If data at current stream position is not available anymore in the receive buffer, the function returns the oldest available line and moves the stream position just after. If no new full line is received, the function returns an empty line.

Returns:
- a string with a single line of text
- On failure, throws an exception or returns a negative error code.
serialport→readMessages()
serialport.readMessages()

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

List<string> readMessages(string pattern, int maxWait)

This function can only be used when the serial port is configured for a message protocol, such as 'Line' mode or MODBUS protocols. It does not work in plain stream modes, eg. 'Char' or 'Byte', for which there is no "start" of message.

The search returns all messages matching the expression provided as argument in the buffer. If no matching message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

Parameters:

- **pattern**: a limited regular expression describing the expected message format, or an empty string if all messages should be returned (no filtering). When using binary protocols, the format applies to the hexadecimal representation of the message.
- **maxWait**: the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

Returns:

- an array of strings containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.
3. Reference

serialport → readStr()

serialport.readStr()

YSerialPort

Reads data from the receive buffer as a string, starting at current stream position.

```
string readStr(int nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

**Parameters:**

- `nChars` the maximum number of characters to read

**Returns:**

- a string with receive buffer contents

On failure, throws an exception or returns a negative error code.
serialport→read_seek() \texttt{serialport.read\_seek()}

Changes the current internal stream position to the specified value.

\texttt{int read\_seek( int rxCountVal)}

This function does not affect the device, it only changes the value stored in the YSerialPort object for the next read operations.

**Parameters :**

- **rxCountVal** the absolute position index (value of rxCount) for next read operations.

**Returns :**

- nothing.
serialport\rightarrow\texttt{registerValueCallback()}

\texttt{serialport.registerValueCallback()}

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

\begin{verbatim}
int registerValueCallback( ValueCallback callback)
\end{verbatim}

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

\textbf{Parameters :}

- \texttt{callback} the callback 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.
Clears the serial port buffer and resets counters to zero.

**int reset()**

Returns:

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Manually sets the state of the RTS line.

```plaintext
int set_RTS(int val)
```

This function has no effect when hardware handshake is enabled, as the RTS line is driven automatically.

**Parameters:**

- `val` 1 to turn RTS on, 0 to turn RTS off

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the logical name of the serial port.

```c
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 serial port.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
Changes the type of protocol used over the serial line.

```c
int set_protocol( string newval)
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

**Parameters :**
- **newval** a string corresponding to the type of protocol used over the serial line

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

On failure, throws an exception or returns a negative error code.
Changes the serial port communication parameters, with a string such as "9600,8N1".

```c
int set_serialMode(string newval)
```

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix can be added to enable flow control: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

**Parameters:**
- `newval` a string corresponding to the serial port communication parameters, with a string such as "9600,8N1"

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

On failure, throws an exception or returns a negative error code.
Stores a user context provided as argument in the userData attribute of the function.

```csharp
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
serialport.writeArray() \[serialport.writeArray()\]

Sends a byte sequence (provided as a list of bytes) to the serial port.

```c
int writeArray( List<int> byteList)
```

**Parameters:**

- `byteList`: a list of byte codes

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Sends a binary buffer to the serial port, as is.

**Parameters:**
- **buff** the binary buffer to send

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

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

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

```c
int writeHex( string hexString);
```

**Parameters:**
- `hexString` a string of hexadecimal byte codes

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

On failure, throws an exception or returns a negative error code.
serialport\rightarrow \text{writeLine}(\text{serialport}.\text{writeLine})

Sends an ASCII string to the serial port, followed by a line break (CR LF).

\begin{verbatim}
int writeLine( string text)
\end{verbatim}

\begin{itemize}
  \item **Parameters** :
    \begin{itemize}
      \item \texttt{text} the text string to send
    \end{itemize}
  \item **Returns** :
    \begin{itemize}
      \item \texttt{YAPI_SUCCESS} if the call succeeds.
    \end{itemize}
\end{itemize}

On failure, throws an exception or returns a negative error code.
serialport.writeMODBUS() Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

```c
int writeMODBUS( string hexString)
```

The message must start with the slave address. The MODBUS CRC/LRC is automatically added by the function. This function does not wait for a reply.

**Parameters:**
- `hexString` a hexadecimal message string, including device address but no CRC/LRC

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

On failure, throws an exception or returns a negative error code.
Sends an ASCII string to the serial port, as is.

```c
int writeStr( string text)
```

**Parameters:**
- `text` the text string to send

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

On failure, throws an exception or returns a negative error code.
3.41. 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 the same move.

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

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

### Global functions

- **yFindServo(func)**
  
  Retrieves a servo for a given identifier.

- **yFirstServo()**
  
  Starts the enumeration of servos currently accessible.

### YServo methods

- **servo→describe()**
  
  Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME) = SERIAL.FUNCTIONID`.

- **servo→get_advertisedValue()**
  
  Returns the current value of the servo (no more than 6 characters).

- **servo→get_enabled()**
  
  Returns the state of the servos.

- **servo→get_enabledAtPowerOn()**
  
  Returns the servo signal generator state at power up.

- **servo→get_errorMessage()**
  
  Returns the error message of the latest error with the servo.

- **servo→get_errorType()**
  
  Returns the numerical error code of the latest error with the servo.

- **servo→get_friendlyName()**
  
  Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.

- **servo→get_functionDescriptor()**
  
  Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **servo→get_functionId()**
  
  Returns the hardware identifier of the servo, without reference to the module.

- **servo→get_hardwareId()**
  
  Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.

- **servo→get_logicalName()**
  
  Returns the logical name of the servo.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servo→get_module()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>servo→get_module_async(callback, context)</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>servo→get_neutral()</td>
<td>Returns the duration in microseconds of a neutral pulse for the servo.</td>
</tr>
<tr>
<td>servo→get_position()</td>
<td>Returns the current servo position.</td>
</tr>
<tr>
<td>servo→get_positionAtPowerOn()</td>
<td>Returns the servo position at device power up.</td>
</tr>
<tr>
<td>servo→get_range()</td>
<td>Returns the current range of use of the servo.</td>
</tr>
<tr>
<td>servo→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>servo→isOnline()</td>
<td>Checks if the servo is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>servo→isOnline_async(callback, context)</td>
<td>Checks if the servo is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>servo→load(msValidity)</td>
<td>Preloads the servo cache with a specified validity duration.</td>
</tr>
<tr>
<td>servo→load_async(msValidity, callback, context)</td>
<td>Preloads the servo cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>servo→move(target, ms_duration)</td>
<td>Performs a smooth move at constant speed toward a given position.</td>
</tr>
<tr>
<td>servo→nextServo()</td>
<td>Continues the enumeration of servos started using yFirstServo().</td>
</tr>
<tr>
<td>servo→registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>servo→set_enabled(newval)</td>
<td>Stops or starts the servo.</td>
</tr>
<tr>
<td>servo→set_enabledAtPowerOn(newval)</td>
<td>Configure the servo signal generator state at power up.</td>
</tr>
<tr>
<td>servo→set_logicalName(newval)</td>
<td>Changes the logical name of the servo.</td>
</tr>
<tr>
<td>servo→set_neutral(newval)</td>
<td>Changes the duration of the pulse corresponding to the neutral position of the servo.</td>
</tr>
<tr>
<td>servo→set_position(newval)</td>
<td>Changes immediately the servo driving position.</td>
</tr>
<tr>
<td>servo→set_positionAtPowerOn(newval)</td>
<td>Configure the servo position at device power up.</td>
</tr>
<tr>
<td>servo→set_range(newval)</td>
<td>Changes the range of use of the servo, specified in per cents.</td>
</tr>
<tr>
<td>servo→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>servo→wait_async(callback, context)</td>
<td>Performs a smooth move at constant speed toward a given position.</td>
</tr>
</tbody>
</table>
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YServo.FindServo()  
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()  
*yFirstServo() YServo.FirstServo()

Starts the enumeration of servos currently accessible.

Use the method `YServo.nextServo()` to iterate on next servos.

**Returns:**

A pointer to a `YServo` object, corresponding to the first servo currently online, or a null pointer if there are none.
servo.describe()

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

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

Returns:

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

**servo**→**get_enabled()**

**servo**→**enabled()** servo.get_enabled()

Returns the state of the servos.

```c
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()
servo
→
enabledAtPowerOn()
servo.get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

<table>
<thead>
<tr>
<th>int get_enabledAtPowerOn()</th>
</tr>
</thead>
</table>

**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.
Returns the error message of the latest error with the servo.

\[
\text{string } \text{get\_errorMessage( )}
\]

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()

servo → errorType()
servo.get_errorType()

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

YRETCODE 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
3. Reference

**servo**→**get_friendlyName()**  \hspace{1cm} **YServo**

**servo**→**friendlyName()**\hspace{1cm}**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()
servo→functionDescriptor()
servo.get_functionDescriptor()

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

`YFUN_DESCR 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`. 
YServo

servo→get_functionId()
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`. 
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.
3. Reference

**servo** → `get_logicalName()`

**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`. 
\textbf{servo\rightarrow\text{get\_module}()}
\textbf{servo\rightarrow\text{module}()\text{servo.get\_module}()}

Gets the \texttt{YModule} object for the device on which the function is located.

\begin{tabular}{|l|}
\hline
\textbf{YModule} \texttt{get\_module()} \\
\hline
\end{tabular}

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

\begin{itemize}
\item \textbf{Returns}:
\begin{itemize}
\item an instance of \texttt{YModule}
\end{itemize}
\end{itemize}
Returns the duration in microseconds of a neutral pulse for the servo.

```c
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`.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servo→get_position()</td>
<td>Returns the current servo position.</td>
</tr>
<tr>
<td>servo→position()servo.get_position()</td>
<td></td>
</tr>
</tbody>
</table>

**int get_position()**

**Returns:**
- an integer corresponding to the current servo position

On failure, throws an exception or returns Y_POSITION_INVALID.
Returns the servo position at device power up.

```c
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`. 

---

3. Reference

YServo

```c
servo->get_positionAtPowerOn()

servo->positionAtPowerOn()

servo.get_positionAtPowerOn()
```

Returns the servo position at device power up.

```c
int get_positionAtPowerOn()
```
servo→get_range()
servo→range() servo.get_range()

Returns the current range of use of the servo.

int get_range()

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>an integer corresponding to the current range of use of the servo</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_RANGE_INVALID.
servo→get_userData()
servo→userData() servo.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.
servo→isOnline() servo.isOnline()  

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

bool 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
Preloads the servo cache with a specified validity duration.

By default, whenever accessing a device, all function attributes are kept in cache for the standard 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()**

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

```c
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.
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()

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

`int registerValueCallback(ValueCallback callback)`

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

**Parameters:**

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

Stops or starts the servo.

```cpp
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.
Configure the servo signal generator state at power up.

```c
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.
Changes the logical name of the servo.

```c
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.
Changes the duration of the pulse corresponding to the neutral position of the servo.

```c
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.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code> an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code> if the call succeeds.</td>
</tr>
</tbody>
</table>

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

**servo** → **set_position()**

Changes immediately the servo driving position.

```c
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→setPositionAtPowerOn()
servo→setPositionAtPowerOn()
servo.setPositionAtPowerOn()

Configure the servo position at device power up.

```c
int setPositionAtPowerOn( 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.
Changes the range of use of the servo, specified in per cents.

```c
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.
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. 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:

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

```nodejs
var yoctolib = require('yoctolib');
var YTemperature = yoctolib.YTemperature;
require_once('yocto_temperature.php');
```

```cpp
#include "yocto_temperature.h"
```

```m
#include "yocto_temperature.h"
```

```pas
uses yocto_temperature;
```

```vb
import com.yoctopuce.YoctoAPI.YTemperature;
```

```cs
import com.yoctopuce.YoctoAPI.YTemperature;
```

```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, in Celsius, as a floating point number.

- `temperature→get_currentValue()`: Returns the current value of the temperature, in Celsius, as a floating point number.

- `temperature→get_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`. 
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature→get_highestValue()</td>
<td>Returns the maximal value observed for the temperature since the device was started.</td>
</tr>
<tr>
<td>temperature→get_logFrequency()</td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td>temperature→get_logicalName()</td>
<td>Returns the logical name of the temperature sensor.</td>
</tr>
<tr>
<td>temperature→get_lowestValue()</td>
<td>Returns the minimal value observed for the temperature since the device was started.</td>
</tr>
<tr>
<td>temperature→get_module()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>temperature→get_module_async(callback, context)</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>temperature→get_recordedData(startTime, endTime)</td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td>temperature→get_reportFrequency()</td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td>temperature→get_resolution()</td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td>temperature→get_sensorType()</td>
<td>Returns the temperature sensor type.</td>
</tr>
<tr>
<td>temperature→get_unit()</td>
<td>Returns the measuring unit for the temperature.</td>
</tr>
<tr>
<td>temperature→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>temperature→isOnline()</td>
<td>Checks if the temperature sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>temperature→isOnline_async(callback, context)</td>
<td>Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>temperature→load(msValidity)</td>
<td>Preloads the temperature sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td>temperature→loadCalibrationPoints(rawValues, refValues)</td>
<td>Retrieves error correction data points previously entered using the method calibrateFromPoints.</td>
</tr>
<tr>
<td>temperature→load_async(msValidity, callback, context)</td>
<td>Preloads the temperature sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>temperature→nextTemperature()</td>
<td>Continues the enumeration of temperature sensors started using yFirstTemperature().</td>
</tr>
<tr>
<td>temperature→registerTimedReportCallback(callback)</td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td>temperature→registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>temperature→set_highestValue(newval)</td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td>temperature→set_logFrequency(newval)</td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
</tbody>
</table>
3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temperature-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the temperature sensor.</td>
</tr>
<tr>
<td><code>temperature-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>temperature-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>temperature-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>temperature-&gt;set_sensorType(newval)</code></td>
<td>Modify the temperature sensor type.</td>
</tr>
<tr>
<td><code>temperature-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>temperature-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
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.FirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

Use the method YTemperature.nextTemperature() to iterate on next temperature sensors.

**Returns:**
- a pointer to a YTemperature object, corresponding to the first temperature sensor currently online, or a null pointer if there are none.
YTemperature
calibrateFromPoints()

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

```c
int calibrateFromPoints( List<double> rawValues,
                        List<double> refValues)
```

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

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

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

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

On failure, throws an exception or returns a negative error code.
temperature.describe() Returns a short text that describes unambiguously the instance of the temperature sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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

Returns:

a string that describes the temperature sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
temperature → get_advertisedValue()  
YTemperature  

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()  \( \text{YTemperature} \)

\[ \text{temperature→currentRawValue()} \]

\[ \text{temperature.get_currentRawValue()} \]

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

\[
\text{double get_currentRawValue( )}
\]

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

On failure, throws an exception or returns \( \text{Y_CURRENTRAWVALUE_INVALID} \).
YTemperature
temperature→get_currentValue()
temperature→currentValue()
temperature.get_currentValue()

Returns the current value of the temperature, in Celsius, as a floating point number.

```java
double get_currentValue() {
    // Implementation
}
```

Returns:
- a floating point number corresponding to the current value of the temperature, in Celsius, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temperature.get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the temperature sensor.</td>
</tr>
<tr>
<td><code>temperature.errorMessage()</code></td>
<td>This method is mostly useful when using the Yoctopuce library with exceptions disabled.</td>
</tr>
<tr>
<td><code>temperature.get_errorMessage()</code></td>
<td>Returns :</td>
</tr>
<tr>
<td></td>
<td>a string corresponding to the latest error message that occurred while using the temperature sensor object</td>
</tr>
</tbody>
</table>
YTemperature

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

YRETCODE 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
YTemperature

\[ \text{temperature} \rightarrow \text{get\_friendlyName()} \]
\[ \text{temperature} \rightarrow \text{friendlyName()} \]
\[ \text{temperature} \text{.get\_friendlyName()} \]

Returns a global identifier of the temperature sensor in the format
\[ \text{MODULE\_NAME.FUNCTION\_NAME} \].

string \text{get\_friendlyName()} \()

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

\begin{itemize}
  \item Returns :
  \begin{itemize}
    \item a string that uniquely identifies the temperature sensor using logical names (ex: MyCustomName.relay1)
  \end{itemize}
  \item On failure, throws an exception or returns \text{Y\_FRIENDLYNAME\_INVALID}.
\end{itemize}
YTemperature
temperature\rightarrow get\_functionDescriptor()
temperature\rightarrow functionDescriptor()
temperature.get\_functionDescriptor()

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

```
YFUN\_DESCR 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\_FUNCTIONDESCRIPTION\_INVALID.
YTemperature

3. Reference

temperature→get_functionId()
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.
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.
3. Reference

**temperature**\\rightarrow**get_highestValue()**

**YTemperature**

**temperature**\\rightarrow**highestValue()**

**temperature.get_highestValue()**

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

```java
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**.
```plaintext
3. Reference

\texttt{YTemperature} \rightarrow \texttt{get\_logFrequency()}

\texttt{YTemperature} \rightarrow \texttt{logFrequency()}

\texttt{YTemperature}.\texttt{get\_logFrequency()}

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

\textbf{string get\_logFrequency()}

\textbf{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 \texttt{Y\_LOGFREQUENCY\_INVALID}.```
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\rightarrow get\_lowestValue()

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

\textbf{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 \texttt{Y\_LOWESTVALUE\_INVALID}.
YTemperature

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 online.

Returns:

an instance of YModule
3. Reference

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

\[
\text{YDataSet \texttt{get\_recordedData}(long startTime, long endTime)}
\]

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 measure, 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 measure, 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.
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**.
Returns the resolution of the measured values.

```c
double get_resolution() // YTemperature
```

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`. 
YTemperature

3. Reference

Returns the temperature sensor type.

```c
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()
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.
3. Reference

YTemperature

\texttt{temperature\rightarrow get\_userData()}
\texttt{temperature\rightarrow userData()}
\texttt{temperature.get\_userData()}

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

\textbf{object} \texttt{get\_userData()}

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

\textbf{Returns}:

the object stored previously by the caller.
Checks if the temperature sensor is currently reachable, without raising any error.

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
Preloads the temperature sensor cache with a specified validity duration.

**YRETCODE load( int 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.
### loadCalibrationPoints

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

```c
int loadCalibrationPoints(List<double> rawValues, List<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.
Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

**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()

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

```c
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.
Register the callback function that is invoked on every change of advertised value.

```c
#include "yTemperature.h"

YTemperature* temperature;

// Register the callback function

void temperatureRegisterValueCallback()
{
    temperature->registerValueCallback();
}
```

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

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
temperature->set_highestValue()  
YTemperature

Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
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.
Changes the logical name of the temperature sensor.

```c
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.
Changes the recorded minimal value observed.

**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.
Changes the timed value notification frequency for this function.

```c
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.
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.
Modify the temperature sensor type.

```c
int set_sensorType( int newval)
```

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

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
3.43. 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:

```
<script type='text/javascript' src='yocto_tilt.js'></script>
```

```
var yoctolib = require('yoctolib');
var YTilt = yoctolib.YTilt;
```

```
require_once('yocto_tilt.php');
```

```
#include "yocto_tilt.h"
```

```
uses yocto_tilt;
```

```
yocto_tilt.vb
```

```
yocto_tilt.cs
```

```
import com.yoctopuce.YoctoAPI.YTilt;
```

```
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, in degrees, as a floating point number.

`tilt → get_currentValue()`
- Returns the current value of the inclination, in degrees, as a floating point number.

`tilt → get_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`. 

---
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_highestValue()</code></td>
<td>Returns the maximal value observed for the inclination since the device was started.</td>
</tr>
<tr>
<td><code>get_logFrequency()</code></td>
<td>Returns the datalogger recording frequency for this function, or &quot;OFF&quot; when measures are not stored in the data logger flash memory.</td>
</tr>
<tr>
<td><code>get_logicalName()</code></td>
<td>Returns the logical name of the tilt sensor.</td>
</tr>
<tr>
<td><code>get_lowestValue()</code></td>
<td>Returns the minimal value observed for the inclination since the device was started.</td>
</tr>
<tr>
<td><code>get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>get_recordedData(startTime, endTime)</code></td>
<td>Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.</td>
</tr>
<tr>
<td><code>get_reportFrequency()</code></td>
<td>Returns the timed value notification frequency, or &quot;OFF&quot; if timed value notifications are disabled for this function.</td>
</tr>
<tr>
<td><code>get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>get_unit()</code></td>
<td>Returns the measuring unit for the inclination.</td>
</tr>
<tr>
<td><code>get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the tilt sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the tilt sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the tilt sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextTilt()</code></td>
<td>Continues the enumeration of tilt sensors started using <code>yFirstTilt()</code>.</td>
</tr>
<tr>
<td><code>registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td>Changes the logical name of the tilt sensor.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>tilt-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>tilt-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>tilt-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>tilt-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>tilt-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
**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()

Starts the enumeration of tilt sensors currently accessible.

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.
3. Reference

**tilt→calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```c
int calibrateFromPoints(List<double> rawValues, List<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

- **rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- **refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

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(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:
a string that describes the tilt sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
Returns the current value of the tilt sensor (no more than 6 characters).

```plaintext
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.
}
```
### Ytilt

#### double get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

**Returns:**

A floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

On failure, throws an exception or returns **Y_CURRENTRAWVALUE_INVALID**.
### 3. Reference

**YTilt**

Returns the current value of the inclination, in degrees, as a floating point number.

```plaintext
double get_currentValue()
```

**Returns:**

- A floating point number corresponding to the current value of the inclination, in degrees, as a floating point number.

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
### YTilt

#### 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.
Returns the numerical error code of the latest error with the tilt sensor.

**YRETCODE 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
Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```plaintext
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 example: `MyCustomName.relay1`)

**Returns:**
- a string that uniquely identifies the tilt sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
**Reference**

`YTilt`  
`tilt.get_functionDescriptor()`  

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```plaintext
YFUN_DESCR 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`**.
3. Reference

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.
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`. 
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**.
3. Reference

```
tilt→get_logFrequency()
```

`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.

```java
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`. 
Returns the logical name of the tilt sensor.

```cpp
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`. 
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.
YModule get_module()

Gets the YModule object for the device on which the function is located.

Returns:

an instance of YModule

If the function cannot be located on any module, the returned instance of YModule is not shown as online.
YTilt

YTilt

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 measure, 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 measure, 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.
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.
Returns the resolution of the measured values.

```java
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`.  

```java
<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the resolution of the measured values</td>
<td>Returns the resolution of the measured values.</td>
</tr>
</tbody>
</table>
```
Returns the measuring unit for the inclination.

```c
string get_unit() 
```

**Returns:**
a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns `Y_UNIT_INVALID`. 
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.
3. Reference

`YTilt`

`tilt.isOnline()`

Checks if the tilt sensor is currently reachable, without raising any error.

```cpp
bool 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.
Preloads the tilt sensor cache with a specified validity duration.

**YRETCODE load( int 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.
3. Reference

**YTilt**

```csharp
tilt.loadCalibrationPoints()
```

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```csharp
int loadCalibrationPoints(List<double> rawValues, List<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.
Continues the enumeration of tilt sensors started using \texttt{yFirstTilt()}.\smallskip

\begin{tabular}{|l|}
\hline
\texttt{YTilt \texttt{nextTilt}()} \\
\hline
\end{tabular}

\textbf{Returns}:

\begin{itemize}
\item \texttt{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.}
\end{itemize}
 yt -> yt.registerTimedReportCallback()
 yt.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.
Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback 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.
### Function: `set_highestValue` (Ytilt)

**Description:**
Changes the recorded maximal value observed.

**Signature:**

```c
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.
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.
Changes the logical name of the tilt sensor.

```c
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.
### 3. Reference

<table>
<thead>
<tr>
<th>tilt→set_lowestValue()</th>
<th>YTilt tilt→setLowestValue() tilt.set_lowestValue()</th>
</tr>
</thead>
</table>

Changes the recorded minimal value observed.

```c
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.
Changes the timed value notification frequency for this function.

```c
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.
Changes the resolution of the measured physical values.

```c
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()

Stores a user context provided as argument in the userData attribute of the function.

```cpp
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
# 3.44. 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:

```javascript
<script type='text/javascript' src='yocto_voc.js'></script>
```

```nodejs
var yoctolib = require('yoctolib');
var YVoc = yoctolib.YVoc;
```

```php
require_once('yocto_voc.php');
```

```cpp
#include "yocto_voc.h"
```

```m
#include "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 \texttt{TYPE\,(NAME)\,=\,SERIAL\,\,FUNCTIONID}.

### `voc→get_advertisedValue()`

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

### `voc→get_currentRawValue()`

Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

### `voc→get_currentValue()`

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

### `voc→get_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 \texttt{MODULE\,\,NAME\,\,FUNCTION\,\,NAME}.

### `voc→get_functionDescriptor()`

Returns a unique identifier of type \texttt{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 since the device was started.

```voc```
get_logFrequency()
```
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```voc```
get_logicalName()
```
Returns the logical name of the Volatile Organic Compound sensor.

```voc```
get_lowestValue()
```
Returns the minimal value observed for the estimated VOC concentration since the device was started.

```voc```
get_module()
```
Gets the YModule object for the device on which the function is located.

```voc```
get_module_async(callback, context)
```
Gets the YModule object for the device on which the function is located (asynchronous version).

```voc```
get_recordedData(startTime, endTime)
```
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```voc```
get_reportFrequency()
```
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```voc```
get_resolution()
```
Returns the resolution of the measured values.

```voc```
get_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```
extVoc()
```
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.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>voc→set_logFrequency(newval)</td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td>voc→set_logicalName(newval)</td>
<td>Changes the logical name of the Volatile Organic Compound sensor.</td>
</tr>
<tr>
<td>voc→set_lowestValue(newval)</td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td>voc→set_reportFrequency(newval)</td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td>voc→set_resolution(newval)</td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td>voc→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>voc→wait_async(callback, context)</td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
3. Reference

**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.FirstVoc()**

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

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\rightarrow calibrateFromPoints()\texttt{voc.calibrateFromPoints()}

Configure error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```c
int calibrateFromPoints( List\langle double\rangle rawValues, List\langle double\rangle refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form \textbf{TYPE (NAME) = SERIAL.FUNCTIONID}.

\textbf{string describe()}

More precisely, \textbf{TYPE} is the type of the function, \textbf{NAME} is the name used for the first access to the function, \textbf{SERIAL} is the serial number of the module if the module is connected or "unresolved", and \textbf{FUNCTIONID} is the hardware identifier of the function if the module is connected. For example, this method returns \textbf{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1} if the module is already connected or \textbf{Relay(BadCustomeName.relay1)=unresolved} if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

\textbf{Returns :}

\textbf{a string that describes the Volatile Organic Compound sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)}
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.
Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

**double get_currentRawValue()**

Returns:

- a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
YVoc

voc.get_currentValue()

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

double get_currentValue()

Returns:
- a floating point number corresponding to the current value of the estimated VOC concentration, in ppm (vol), as a floating point number

On failure, throws an exception or returns Y CURRENTVALUE INVALID.
3. Reference

YVoc

voc→getErrorMessage()

voc→errorMessage() voc.getErrorMessage()

YVoc

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
Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

**YRETCODE 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
Returns a global identifier of the Volatile Organic Compound sensor in the format
`MODULE_NAME.FUNCTION_NAME`.

```c
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 example: `MyCustomName.relay1`)

**Returns**:  
a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex:  
`MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 

```
voc→get_functionDescriptor()
voc→functionDescriptor()
voc.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR 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.
<table>
<thead>
<tr>
<th>Returns:</th>
<th>a string that identifies the Volatile Organic Compound sensor (ex: relay1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>On failure,</td>
<td>throws an exception or returns Y_FUNCTIONID_INVALID.</td>
</tr>
</tbody>
</table>

3. Reference

\[ \text{voc.get\_functionId()} \]

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

**Example:**

```java
string get_functionId() {
    // Implementation
}
```

For example, `relay1`
Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form SERIAL.FUNCTIONID.

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.
Returns the maximal value observed for the estimated VOC concentration since the device was started.

**Returns:**
- A floating point number corresponding to the maximal value observed for the estimated VOC concentration since the device was started.

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`. 
voc→get_logFrequency()
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.
YVoc

voc.get_logicalName()

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 since the device was started.

double get_lowestValue()

**Returns:**

a floating point number corresponding to the minimal value observed for the estimated VOC concentration since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
voc→get_module()

```
YModule get_module()
```

Gets the YModule object for the device on which the function is located.

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.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startTime</strong></td>
</tr>
<tr>
<td>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 measure, without initial limit.</td>
</tr>
</tbody>
</table>

| **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 measure, without ending limit. |

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.</td>
</tr>
</tbody>
</table>
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

Returns:
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.
voc→get_resolution()
voc→resolution()voc.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()
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()

voc→userData()voc.get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

object get_userData()

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns:

the object stored previously by the caller.
YVoc

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

**bool 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() \( \rightarrow \) voc.load()

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

**YRETCODE load( int 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()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```c
int loadCalibrationPoints( List<double> rawValues, List<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()

Continues the enumeration of Volatile Organic Compound sensors started using yFirstVoc().

YVoc

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.
3. Reference

voc→registerTimedReportCallback()

voc.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

```c
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()

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback 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.
Changes the recorded maximal value observed.

```c
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.
Changes the datalogger recording frequency for this function.

```c
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.

```cpp
int setLogicalName( 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\rightarrow set\_lowestValue() \\
voc\rightarrow setLowestValue() \\
voc.set\_lowestValue()

Changes the recorded minimal value observed.

\textbf{int set\_lowestValue( double newval)}

\textbf{Parameters :}

newval a floating point number corresponding to the recorded minimal value observed

\textbf{Returns :}

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
3. Reference

```c
voc->set_reportFrequency()
voc->setReportFrequency()
voc.set_reportFrequency()
```

Changes the timed value notification frequency for this function.

```c
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".

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code> a string corresponding to the timed value notification frequency for this function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>YAPI_SUCCESS</code> if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
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.
Stores a user context provided as argument in the userData attribute of the function.

```cpp
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
3.45. 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:

<table>
<thead>
<tr>
<th>js</th>
<th>&lt;script type='text/javascript' src='yocto_voltage.js'&gt;&lt;/script&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodejs</td>
<td>var yoctolib = require('yoctolib');</td>
</tr>
<tr>
<td></td>
<td>var YVoltage = yoctolib.YVoltage;</td>
</tr>
<tr>
<td>php</td>
<td>require_once('yocto_voltage.php');</td>
</tr>
<tr>
<td>cpp</td>
<td>#include &quot;yocto_voltage.h&quot;</td>
</tr>
<tr>
<td>m</td>
<td>#import &quot;yocto_voltage.h&quot;</td>
</tr>
<tr>
<td>pas</td>
<td>uses yocto_voltage;</td>
</tr>
<tr>
<td>vb</td>
<td>yocto_voltage.vb</td>
</tr>
<tr>
<td>cs</td>
<td>yocto_voltage.cs</td>
</tr>
<tr>
<td>java</td>
<td>import com.yoctopuce.YoctoAPI.YVoltage;</td>
</tr>
<tr>
<td>py</td>
<td>from yocto_voltage import *</td>
</tr>
</tbody>
</table>

### 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, in Volt, as a floating point number.

**voltage→get_currentValue()**

Returns the current value of the voltage, in Volt, as a floating point number.

**voltage→get_errorMessage()**

Returns the error message of the latest error with the voltage sensor.

**voltage→get_errorType()**

Returns the numerical error code of the latest error with the voltage sensor.

**voltage→get_friendlyName()**

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

**voltage→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**voltage→get_functionId()**

Returns the hardware identifier of the voltage sensor, without reference to the module.

**voltage→get_hardwareId()**

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`. 
voltage→get_highestValue()
  Returns the maximal value observed for the voltage since the device was started.

voltage→get_logFrequency()
  Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the
data logger flash memory.

voltage→get_logicalName()
  Returns the logical name of the voltage sensor.

voltage→get_lowestValue()
  Returns the minimal value observed for the voltage since the device was started.

voltage→get_module()
  Gets the YModule object for the device on which the function is located.

voltage→get_module_async(callback, context)
  Gets the YModule object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(startTime, endTime)
  Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()
  Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()
  Returns the resolution of the measured values.

voltage→get_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.

voltage→set_logFrequency(newval)
  Changes the datalogger recording frequency for this function.

voltage→set_logicalName(newval)
  Changes the logical name of the voltage sensor.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>voltage-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>voltage-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>voltage-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>voltage-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>voltage-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YVoltage.FindVoltage() YVoltage.FindVoltage()

Retrieves a voltage sensor for a given identifier.

```go
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()  
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.
3. Reference

YVoltage.voltage.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```c
int calibrateFromPoints( List<double> rawValues, List<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
voltage→describe()"voltage.describe()

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
dunction, SERIAL is the serial number of the module if the module is connected or "unresolved",
and FUNCTIONID is the hardware identifier of the function if the module is connected. For example,
this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the
module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module
has not yet been connected. This method does not trigger any USB or TCP transaction and can
therefore be used in a debugger.

Returns :
a string that describes the voltage sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
voltage \rightarrow \text{get\_advertisedValue()}

voltage \rightarrow \text{advertisedValue()}

voltage.get\_advertisedValue()

\text{YVoltage}

Returns the current value of the voltage sensor (no more than 6 characters).

\begin{verbatim}
string get\_advertisedValue() 
\end{verbatim}

\text{Returns :}

\begin{itemize}
  \item a string corresponding to the current value of the voltage sensor (no more than 6 characters).
\end{itemize}

On failure, throws an exception or returns \text{Y\_ADVERTISEDVALUE\_INVALID}. 
voltage→get_currentRawValue()
voltage→currentRawValue()
voltage.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

double get_currentRawValue()

**Returns:**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
3. Reference

YVoltage

voltage.get_currentValue()

Returns the current value of the voltage, in Volt, as a floating point number.

double get_currentValue()

Returns:

- a floating point number corresponding to the current value of the voltage, in Volt, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
voltage\rightarrow \text{get\_error\_message()}

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

\begin{tabular}{|m{10cm}|}
\hline
\textbf{Returns :} \\
a string corresponding to the latest error message that occurred while using the voltage sensor object \\
\hline
\end{tabular}
YVoltage

voltage.get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

YRETCODE 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
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 example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the voltage sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
3. Reference

YVoltage
voltage \rightarrow \text{get\_functionDescriptor()}

YVoltage
voltage \rightarrow \text{functionDescriptor()}

voltage.get\_functionDescriptor()

Returns a unique identifier of type \text{YFUN\_DESCR} corresponding to the function.

\text{YFUN\_DESCR get\_functionDescriptor( )}

This identifier can be used to test if two instances of \text{YFunction} reference the same physical function on the same physical device.

\textbf{Returns :}

an identifier of type \text{YFUN\_DESCR}.

\textbf{If the function has never been contacted, the returned value is} \text{Y\_FUNCTIONDESCRIPTOR\_INVALID}. 
voltage→get_functionId()

voltage→functionId() voltage.get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

```cpp
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`. 
Returns the unique hardware identifier of the voltage sensor in the form `SERIALFUNCTIONID`.

```c
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\rightarrow \text{get\_highestValue()}
voltage\rightarrow \text{highestValue()}\text{voltage.get\_highestValue()}

Returns the maximal value observed for the voltage since the device was started.

\textbf{double get\_highestValue()}

\textbf{Returns :}

a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns \texttt{Y\_HIGHESTVALUE\_INVALID}.
>Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

\texttt{string get\_logFrequency()}\hfill \texttt{YVoltage}

\begin{tabular}[t]{|p{1\textwidth}|}
\hline
\textbf{Returns :} \\
\quad a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory \\
\hline
\text{On failure, throws an exception or returns } \texttt{Y\_LOGFREQUENCY\_INVALID}. \\
\hline
\end{tabular}
<table>
<thead>
<tr>
<th><strong>Returns</strong></th>
<th>Returns the logical name of the voltage sensor.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>string get_logicalName()</strong></td>
<td>Returns a string corresponding to the logical name of the voltage sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.</td>
</tr>
</tbody>
</table>
### 3. Reference

**voltage→get_lowestValue()**

`voltage→lowestValue()`

`voltage.get_lowestValue()`

<table>
<thead>
<tr>
<th>Returns the minimal value observed for the voltage since the device was started.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>double get_lowestValue()</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Returns</strong> :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the minimal value observed for the voltage since the device was started</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`. 
voltage→get_module()  
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 online.

Returns:

an instance of YModule
3. Reference

YVoltage

voltage\rightarrow get\_recordedData()\rightarrow voltage.get\_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

<table>
<thead>
<tr>
<th>YDataSet get_recordedData( long startTime, long endTime)</th>
</tr>
</thead>
</table>

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 measure, 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 measure, 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.
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.
Returns the resolution of the measured values.

def 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\rightarrow get\_unit()
voltage\rightarrow 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.
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**Returns**:  
the object stored previously by the caller.
YVoltage

Voltage → isOnline() → voltage.isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

`bool 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
3. Reference

voltage→load()voltage.load()  YVoltage

Preloads the voltage sensor cache with a specified validity duration.

YRETCODE load( int 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.

<table>
<thead>
<tr>
<th>Parameters :</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>msValidity</strong> an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS when the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
YVoltage

voltage.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```c
int loadCalibrationPoints(List<double> rawValues, List<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()\n
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()

Registers the callback function that is invoked on every periodic timed notification.

```c
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.
Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- **callback** the callback 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.
Changes the recorded maximal value observed.

**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.
Changes the datalogger recording frequency for this function.

```c
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()

Changes the logical name of the voltage sensor.

```c
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()

Changes the recorded minimal value observed.

```c
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.
Changes the timed value notification frequency for this function.

```c
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 (e.g. "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.
Changes the resolution of the measured physical values.

```c
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.
3. Reference

The function `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
3.46. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions.

In order to use the functions described here, you should include:

```js
<script type='text/javascript' src='yocto_vsource.js'></script>
```

```php
require_once('yocto_vsource.php');
```

```cpp
#include "yocto_vsource.h"
```

```m
#import "yocto_vsource.h"
```

```pas
uses yocto_vsource;
```

```vb
yocto_vsource.vb
```

```cs
yocto_vsource.cs
```

```java
import com.yoctopuce.YoctoAPI.YVSource;
```

```py
from yocto_vsource import *
```

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindVSource(func)</code></td>
<td>Retrieves a voltage source for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstVSource()</code></td>
<td>Starts the enumeration of voltage sources currently accessible.</td>
</tr>
</tbody>
</table>

### YVSource methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vsource.describe()</code></td>
<td>Returns a short text that describes the function in the form ( \text{TYPE} \ (\text{NAME}) = \text{SERIAL} \cdot \text{FUNCTIONID} ).</td>
</tr>
<tr>
<td><code>vsource.get_advertisedValue()</code></td>
<td>Returns the current value of the voltage source (no more than 6 characters).</td>
</tr>
<tr>
<td><code>vsource.get_errorMessage()</code></td>
<td>Returns the error message of the latest error with this function.</td>
</tr>
<tr>
<td><code>vsource.get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with this function.</td>
</tr>
<tr>
<td><code>vsource.get_extPowerFailure()</code></td>
<td>Returns true if external power supply voltage is too low.</td>
</tr>
<tr>
<td><code>vsource.get_failure()</code></td>
<td>Returns true if the module is in failure mode.</td>
</tr>
<tr>
<td><code>vsource.get_friendlyName()</code></td>
<td>Returns a global identifier of the function in the format ( \text{MODULE} \cdot \text{FUNCTION_NAME} ).</td>
</tr>
<tr>
<td><code>vsource.get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type ( \text{YFUN} \cdot \text{DESCR} ) corresponding to the function.</td>
</tr>
<tr>
<td><code>vsource.get_functionId()</code></td>
<td>Returns the hardware identifier of the function, without reference to the module.</td>
</tr>
<tr>
<td><code>vsource.get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the function in the form ( \text{SERIAL} \cdot \text{FUNCTIONID} ).</td>
</tr>
<tr>
<td><code>vsource.get_logicalName()</code></td>
<td>Returns the logical name of the voltage source.</td>
</tr>
<tr>
<td><code>vsource.get_module()</code></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>vsource.get_module_async(callback, context)</code></td>
<td>}</td>
</tr>
</tbody>
</table>
Gets the `YModule` object for the device on which the function is located (asynchronous version).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vsource-&gt;get_overCurrent()</code></td>
<td>Returns true if the appliance connected to the device is too greedy.</td>
</tr>
<tr>
<td><code>vsource-&gt;get_overHeat()</code></td>
<td>Returns TRUE if the module is overheating.</td>
</tr>
<tr>
<td><code>vsource-&gt;get_overLoad()</code></td>
<td>Returns true if the device is not able to maintain the requested voltage output.</td>
</tr>
<tr>
<td><code>vsource-&gt;get_regulationFailure()</code></td>
<td>Returns true if the voltage output is too high regarding the requested voltage.</td>
</tr>
<tr>
<td><code>vsource-&gt;get_unit()</code></td>
<td>Returns the measuring unit for the voltage.</td>
</tr>
<tr>
<td><code>vsource-&gt;get_userData()</code></td>
<td>Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><code>vsource-&gt;get_voltage()</code></td>
<td>Returns the voltage output command (mV)</td>
</tr>
<tr>
<td><code>vsource-&gt;isOnline()</code></td>
<td>Checks if the function is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>vsource-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the function is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>vsource-&gt;load(msValidity)</code></td>
<td>Preloads the function cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>vsource-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the function cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>vsource-&gt;nextVSource()</code></td>
<td>Continues the enumeration of voltage sources started using <code>yFirstVSource()</code>.</td>
</tr>
<tr>
<td><code>vsource-&gt;pulse(voltage, ms_duration)</code></td>
<td>Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.</td>
</tr>
<tr>
<td><code>vsource-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>vsource-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the voltage source.</td>
</tr>
<tr>
<td><code>vsource-&gt;set_userData(data)</code></td>
<td>Stores a user context provided as argument in the <code>userData</code> attribute of the function.</td>
</tr>
<tr>
<td><code>vsource-&gt;set_voltage(newval)</code></td>
<td>Tunes the device output voltage (milliVolts).</td>
</tr>
<tr>
<td><code>vsource-&gt;voltageMove(target, ms_duration)</code></td>
<td>Performs a smooth move at constant speed toward a given value.</td>
</tr>
<tr>
<td><code>vsource-&gt;wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
yFindVSource() — YVSource.FindVSource()

Retrieves a voltage source for a given identifier.

YVSource FindVSource( 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 YVSource.isOnline() to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters:

- func a string that uniquely characterizes the voltage source

Returns:

- a YVSource object allowing you to drive the voltage source.
**yFirstVSource() — YVSource.FirstVSource()**

Starts the enumeration of voltage sources currently accessible.

YVSource FirstVSource()

Use the method `YVSource.nextVSource()` to iterate on next voltage sources.

**Returns**:

- a pointer to a `YVSource` object, corresponding to the first voltage source currently online, or a null pointer if there are none.
vsource.describe() Returns a short text that describes the function in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:
- a string that describes the function (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
returns the current value of the voltage source (no more than 6 characters).

string get_advertisedValue()

YVSource

vsourse.get_advertisedValue()

vsourse→advertisedValue()

vsourse→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.
### 3. Reference

**YVSource**

vsourcè→get_errorMessage()

vsourcè→errorMessage()

vsourcè.get_errorMessage()

---

Returns the error message of the latest error with this function.

```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 function object.
Returns the numerical error code of the latest error with this function.

YRETCODE `getErrorType()`

Returns:
- a number corresponding to the code of the latest error that occurred while using this function object.
3. Reference

VSource

YVSource

Returns true if external power supply voltage is too low.

int get_extPowerFailure()

Returns true if external power supply voltage is too low.

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 \rightarrow \text{get\_failure()}

\text{vsource\rightarrow failure()}\text{vsource.get\_failure()}

Returns true if the module is in failure mode.

\text{int get\_failure()}

vsource \rightarrow \text{get\_failure()}

\text{vsource\rightarrow failure()}\text{vsource.get\_failure()}

Returns true if the module is in failure mode.

\begin{verbatim}
js  function get_failure()  
php function get_failure()  
cpp Y_FAILURE_enum get_failure()  
m -(Y_FAILURE_enum) failure  
pas function get_failure(): Integer  
vb function get_failure(): Integer  
java int get_failure()  
py def get_failure()  
\end{verbatim}

More information can be obtained by testing get\_overheat, get\_overcurrent etc... When an error condition is met, the output voltage is set to zéro and cannot be changed until the reset() function is called.

\textbf{Returns}:

either \text{Y\_FAILURE\_FALSE} or \text{Y\_FAILURE\_TRUE}, according to true if the module is in failure mode

On failure, throws an exception or returns \text{Y\_FAILURE\_INVALID}.
3. Reference

vsource→get_friendlyName()
vsource→friendlyName()vsource.get_friendlyName()

Returns a global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

override string get_friendlyName()

vsource→get_friendlyName()
vsource→friendlyName()vsource.get_friendlyName()

Returns a global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

<table>
<thead>
<tr>
<th>js</th>
<th>function get_friendlyName( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>php</td>
<td>function get_friendlyName( )</td>
</tr>
<tr>
<td>cpp</td>
<td>virtual string get_friendlyName( )</td>
</tr>
<tr>
<td>m</td>
<td>-(NSString*) friendlyName</td>
</tr>
<tr>
<td>cs</td>
<td>override string get_friendlyName( )</td>
</tr>
<tr>
<td>java</td>
<td>String get_friendlyName( )</td>
</tr>
</tbody>
</table>

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.
3. Reference

YVSource

YVSource

vsourceref get_functionDescriptor() vsourceref functionDescriptor() vsourceref get_vsourceDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR get_functionDescriptor()

vsourceref get_functionDescriptor() vsourceref functionDescriptor() vsourceref get_vsourceDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| js   | function get_functionDescriptor() |
| php  | function get_functionDescriptor() |
| cpp  | YFUN_DESCR get_functionDescriptor() |
| m    | (YFUN_DESCR) functionDescriptor |
| pas  | function get_functionDescriptor(): YFUN_DESCR |
| vb   | function get_functionDescriptor() As YFUN_DESCR |
| cs   | YFUN_DESCR get_functionDescriptor() |
| java | String get_functionDescriptor() |
| py   | def get_functionDescriptor() |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns:**

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
Returns the hardware identifier of the function, without reference to the module.

```javascript
function get_functionId() {
    // JavaScript code
}
```
```php
function get_functionId() {
    // PHP code
}
```
```cpp
string get_functionId() {
    // C++ code
}
```
```objc
- (NSString*) functionId
```
```vb
function get_functionId() As String
```
```cs
string get_functionId()
```
```java
String get_functionId()
```

For example `relay1`

Returns:

A string that identifies the function (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
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.
3. Reference

YVSource

target get_logicalName

Returns a string corresponding to the logical name of the voltage source.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.


```plaintext
vsourceref_module()  
vsourceref_module()vsourceref_module()

Gets the YModule object for the device on which the function is located.

YModule ref_module()

vsourceref_module()  
vsourceref_module()vsourceref_module()

Gets the YModule object for the device on which the function is located.

js  function ref_module()  
php  function ref_module()  
cpp  YModule * ref_module()  
m   -(YModule*) module  
pas  function ref_module() : TYModule  
vb  function ref_module() As YModule  
java  YModule ref_module()  
py  def ref_module()  

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :
   an instance of YModule
```
vsourse→get_overCurrent()
YVSource
vsourse→overCurrent() vsourse.get_overCurrent()

Returns true if the appliance connected to the device is too greedy.

int get_overCurrent( )

vsourse→get_overCurrent()
vsourse→overCurrent() vsourse.get_overCurrent()

Returns true if the appliance connected to the device is too greedy.

<table>
<thead>
<tr>
<th>js</th>
<th>function get_overCurrent( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>php</td>
<td>function get_overCurrent( )</td>
</tr>
<tr>
<td>cpp</td>
<td>Y_OVERCURRENT_enum get_overCurrent( )</td>
</tr>
<tr>
<td>m</td>
<td>(Y_OVERCURRENT_enum) overCurrent</td>
</tr>
<tr>
<td>pas</td>
<td>function get_overCurrent( ): Integer</td>
</tr>
<tr>
<td>vb</td>
<td>function get_overCurrent( ) As Integer</td>
</tr>
<tr>
<td>cs</td>
<td>int get_overCurrent( )</td>
</tr>
<tr>
<td>java</td>
<td>int get_overCurrent( )</td>
</tr>
<tr>
<td>py</td>
<td>def get_overCurrent( )</td>
</tr>
<tr>
<td>cmd</td>
<td>YVSource target get_overCurrent</td>
</tr>
</tbody>
</table>

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.
vsourse→get_overHeat()
vsourse→overHeat()vsourse.get_overHeat()

Returns TRUE if the module is overheating.

int get_overHeat( )

vsourse→get_overHeat()
vsourse→overHeat()vsourse.get_overHeat()

Returns TRUE if the module is overheating.

<table>
<thead>
<tr>
<th>js</th>
<th>function get_overHeat( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>php</td>
<td>function get_overHeat( )</td>
</tr>
<tr>
<td>cpp</td>
<td>Y_OVERHEAT_enum get_overHeat( )</td>
</tr>
<tr>
<td>m</td>
<td>-(Y_OVERHEAT_enum) overHeat</td>
</tr>
<tr>
<td>pas</td>
<td>function get_overHeat( ): Integer</td>
</tr>
<tr>
<td>vb</td>
<td>function get_overHeat( ) As Integer</td>
</tr>
<tr>
<td>cs</td>
<td>int get_overHeat( )</td>
</tr>
<tr>
<td>java</td>
<td>int get_overHeat( )</td>
</tr>
<tr>
<td>py</td>
<td>def get_overHeat( )</td>
</tr>
<tr>
<td>cmd</td>
<td>YVSource target get_overHeat</td>
</tr>
</tbody>
</table>

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.
3. Reference

YVSource

vsoure→get_overLoad()
vsoure→overLoad() vsoure.get_overLoad()

Returns true if the device is not able to maintain the requested voltage output.

int get_overLoad()

vsoure→get_overLoad()
vsoure→overLoad() vsoure.get_overLoad()

Returns true if the device is not able to maintain the requested voltage output.

| js | function get_overLoad() |
| php | function get_overLoad() |
| cpp | Y_OVERLOAD_enum get_overLoad() |
| m | (Y_OVERLOAD_enum) overLoad |
| pas | function get_overLoad(): Integer |
| vb | function get_overLoad() As Integer |
| cs | int get_overLoad() |
| java | int get_overLoad() |
| py | def get_overLoad() |
| cmd | YVSource target get_overLoad |

Returns :

either Y_OVERLOAD_FALSE or Y_OVERLOAD_TRUE, according to true if the device is not able to maintain the requested voltage output

On failure, throws an exception or returns Y_OVERLOAD_INVALID.
Returns true if the voltage output is too high regarding the requested voltage.

**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`. 
YVSource

3. Reference

- **vsource**→**get_unit()**
- **vsource**→**unit()**
- **vsource.get_unit()**

Returns the measuring unit for the voltage.

- **string get_unit()**

- **vsource**→**get_unit()**
- **vsource**→**unit()**
- **vsource.get_unit()**

Returns the measuring unit for the voltage.

- **js**
  `function get_unit()`
- **php**
  `function get_unit()`
- **cpp**
  `string get_unit();`
- **m**
  `<(NSString*) unit`
- **pas**
  `function get_unit(): string`
- **vb**
  `function get_unit() As String`
- **cs**
  `string get_unit();`
- **java**
  `String get_unit();`
- **py**
  `def get_unit();`
- **cmd**
  `YVSource target get_unit`

**Returns**:
- A string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns **Y_UNIT_INVALID**.
Returns the value of the userData attribute, as previously stored using method `set_userData`.

```markdown
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.
Returns the voltage output command (mV)

```js
function get_voltage() {
  // JavaScript implementation
}
```
```php
function get_voltage() {
  // PHP implementation
}
```
```cpp
int get_voltage() {
  // C++ implementation
}
```
```m
-(int) voltage
```
```pas
function get_voltage( ): LongInt
```
```vb
function get_voltage( ) As Integer
```
```cs
int get_voltage( )
```
```java
int get_voltage( )
```
```py
def get_voltage( )
```

**Returns:**

an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns `Y_VOLTAGE_INVALID`. 
vsoure→isOnline()vsource.isOnline()

Checks if the function is currently reachable, without raising any error.

bool isOnline()

vsoure→isOnline()vsource.isOnline()

Checks if the function is currently reachable, without raising any error.

<table>
<thead>
<tr>
<th>js</th>
<th>function isOnline()</th>
</tr>
</thead>
<tbody>
<tr>
<td>php</td>
<td>function isOnline()</td>
</tr>
<tr>
<td>cpp</td>
<td>bool isOnline()</td>
</tr>
<tr>
<td>m</td>
<td>-(BOOL) isOnline</td>
</tr>
<tr>
<td>pas</td>
<td>function isOnline() : boolean</td>
</tr>
<tr>
<td>vb</td>
<td>function isOnline( ) As Boolean</td>
</tr>
<tr>
<td>cs</td>
<td>bool isOnline( )</td>
</tr>
<tr>
<td>java</td>
<td>boolean isOnline( )</td>
</tr>
<tr>
<td>py</td>
<td>def isOnline( )</td>
</tr>
</tbody>
</table>

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
Preloads the function cache with a specified validity duration.

**YRETCODE load( int 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.
Continues the enumeration of voltage sources started using `yFirstVSource()`.

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>js</td>
<td><code>function nextVSource() {</code></td>
</tr>
<tr>
<td>php</td>
<td><code>function nextVSource() {</code></td>
</tr>
<tr>
<td>cpp</td>
<td><code>YVSource * nextVSource();</code></td>
</tr>
<tr>
<td>m</td>
<td><code>-(YVSource*) nextVSource</code></td>
</tr>
<tr>
<td>pas</td>
<td><code>function nextVSource( ); TYVSource</code></td>
</tr>
<tr>
<td>vb</td>
<td><code>function nextVSource( ) As YVSource</code></td>
</tr>
<tr>
<td>cs</td>
<td><code>YVSource nextVSource( )</code></td>
</tr>
<tr>
<td>java</td>
<td><code>YVSource nextVSource( )</code></td>
</tr>
<tr>
<td>py</td>
<td><code>def nextVSource( )</code></td>
</tr>
</tbody>
</table>

**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()

Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.

### 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.
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)
```

```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()

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.

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.
Stores a user context provided as argument in the userData attribute of the function.

```javascript
function set_userData( data)
```

```php
function set_userData( $data)
```

```cpp
void set_userData( void* data)
```

```objc```
-(void) setUserData : (void*) data
```

```pas```
procedure set_userData( data: Tobject)
```

```vb```
procedure set_userData( ByVal data As Object)
```

```java```
void set_userData( Object data)
```

```py```
def set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters:**

- `data` any kind of object to be stored
### 3. Reference

<table>
<thead>
<tr>
<th>YVSource</th>
<th>vsource.set_voltage()</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsource.setVoltage()</td>
<td>vsource.set_voltage()</td>
</tr>
</tbody>
</table>

Tunes the device output voltage (milliVolts).

```plaintext
int set_voltage( int newval)
```

**Parameters:**

- `newval` an integer

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
YVSource\texttt{.voltageMove()} \hspace{1cm} \textbf{Performs a smooth move at constant speed toward a given value.}

\begin{verbatim}
int\ vsource.voltageMove(int\ target, int\ ms\_duration)
\end{verbatim}

\begin{verbatim}
YVSource target\ voltageMove target\ ms\_duration
\end{verbatim}

**Parameters:**

- \textbf{target} \hspace{1cm} new output value at end of transition, in milliVolts.
- \textbf{ms\_duration} \hspace{1cm} transition duration, in milliseconds

**Returns:**

- \textbf{YAPI\_SUCCESS} if the call succeeds.

On failure, throws an exception or returns a negative error code.
3.47. 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:

```javascript
<script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
```
```php
require_once('yocto_wakeupmonitor.php');
```
```cpp
#include "yocto_wakeupmonitor.h"
```
```m
#include "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()**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get_powerDuration()</strong></td>
<td>Returns the next scheduled wake up date/time (UNIX format)</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>get_powerDuration()</strong></td>
<td>Returns the maximal wake up time (in seconds) before automatically going to sleep.</td>
</tr>
<tr>
<td><strong>get_sleepCountdown()</strong></td>
<td>Returns the delay before the next sleep period.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>get_sleepCountdown()</strong></td>
<td>Returns the delay before the next sleep period.</td>
</tr>
<tr>
<td><strong>get_userData()</strong></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>get_userData()</strong></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><strong>get_wakeUpReason()</strong></td>
<td>Returns the latest wake up reason.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>get_wakeUpReason()</strong></td>
<td>Returns the latest wake up reason.</td>
</tr>
<tr>
<td><strong>get_wakeUpState()</strong></td>
<td>Returns the current state of the monitor</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>get_wakeUpState()</strong></td>
<td>Returns the current state of the monitor</td>
</tr>
<tr>
<td><strong>isOnline()</strong></td>
<td>Checks if the monitor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>isOnline_async(callback, context)</strong></td>
<td>Checks if the monitor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><strong>load(msValidity)</strong></td>
<td>Preloads the monitor cache with a specified validity duration.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>load_async(msValidity, callback, context)</strong></td>
<td>Preloads the monitor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><strong>nextWakeUpMonitor()</strong></td>
<td>Continues the enumeration of monitors started using yFirstWakeUpMonitor().</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>nextWakeUpMonitor()</strong></td>
<td>Continues the enumeration of monitors started using yFirstWakeUpMonitor().</td>
</tr>
<tr>
<td><strong>registerValueCallback(callback)</strong></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><strong>resetSleepCountDown()</strong></td>
<td>Resets the sleep countdown.</td>
</tr>
<tr>
<td><strong>set_logicalName(newval)</strong></td>
<td>Changes the logical name of the monitor.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>set_logicalName(newval)</strong></td>
<td>Changes the logical name of the monitor.</td>
</tr>
<tr>
<td><strong>set_nextWakeUp(newval)</strong></td>
<td>Changes the days of the week when a wake up must take place.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>set_nextWakeUp(newval)</strong></td>
<td>Changes the days of the week when a wake up must take place.</td>
</tr>
<tr>
<td><strong>set_powerDuration(newval)</strong></td>
<td>Changes the maximal wake up time (seconds) before automatically going to sleep.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>set_powerDuration(newval)</strong></td>
<td>Changes the maximal wake up time (seconds) before automatically going to sleep.</td>
</tr>
<tr>
<td><strong>set_sleepCountdown(newval)</strong></td>
<td>Changes the delay before the next sleep period.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>set_sleepCountdown(newval)</strong></td>
<td>Changes the delay before the next sleep period.</td>
</tr>
<tr>
<td><strong>set_userData(data)</strong></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>sleep(secBeforeSleep)</strong></td>
<td>Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.</td>
</tr>
<tr>
<td><strong>set_userData(data)</strong></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><strong>sleepFor(secUntilWakeUp, secBeforeSleep)</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>wakeupmonitor</strong>→<strong>sleepUntil(wakeUpTime, secBeforeSleep)</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>wait_async(callback, context)</strong></td>
<td></td>
</tr>
</tbody>
</table>
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()

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.
3. Reference

**YWakeUpMonitor.FirstWakeUpMonitor()**

*YWakeUpMonitor.FirstWakeUpMonitor()*

starts the enumeration of monitors currently accessible.

Use the method **YWakeUpMonitor.nextWakeUpMonitor()** to iterate on next monitors.

Returns:

- a pointer to a **YWakeUpMonitor** object, corresponding to the first monitor currently online, or a null pointer if there are none.
YWakeUpMonitor

wakeupmonitor.describe()

Returns a short text that describes unambiguously the instance of the monitor in the form
TYPE(NAME)=SERIAL.FUNCTIONID.

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the
function, SERIAL is the serial number of the module if the module is connected or "unresolved",
and FUNCTIONID is the hardware identifier of the function if the module is connected. For example,
this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the
module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module
has not yet been connected. This method does not trigger any USB or TCP transaction and can
therefore be used in a debugger.

Returns :
 a string that describes the monitor (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
ywakeupmonitor\rightarrow get\_advertisedValue() \quad \text{YWakeupMonitor}

ywakeupmonitor\rightarrow advertisedValue()  

ywakeupmonitor.get\_advertisedValue()  

Returns the current value of the monitor (no more than 6 characters).

\begin{center}
\textbf{string get\_advertisedValue( )}
\end{center}

\begin{itemize}
\item Returns: 
  a string corresponding to the current value of the monitor (no more than 6 characters).
\item On failure, throws an exception or returns \texttt{Y\_ADVERTISEDVALUE\_INVALID}.
\end{itemize}
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.
### 3. Reference

#### YWakeUpMonitor

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the monitor.</td>
</tr>
<tr>
<td><code>errorType()</code></td>
<td>This method is mostly useful when using the Yoctopuce library with exceptions disabled.</td>
</tr>
</tbody>
</table>

**RET CODE `get_errorType()`**

- **Returns**: a number corresponding to the code of the latest error that occurred while using the monitor object.
YWakeUpMonitor

WakeUpMonitor→get_friendlyName()
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 example: MyCustomName.relay1)

Returns:

- a string that uniquely identifies the monitor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
YWakeUpMonitor

wakeupmonitor \rightarrow \text{get\_functionDescriptor()}

\text{wakeupmonitor} \rightarrow \text{functionDescriptor()}

\text{wakeupmonitor.get\_functionDescriptor()}

Returns a unique identifier of type \text{YFUN\_DESCR} corresponding to the function.

\text{YFUN\_DESCR get\_functionDescriptor( )}

This identifier can be used to test if two instances of \text{YFunction} reference the same physical function on the same physical device.

\text{Returns :}

an identifier of type \text{YFUN\_DESCR}.

If the function has never been contacted, the returned value is \text{Y\_FUNCTIONDESCRIPTOR\_INVALID}.
wakeupmonitor→get_functionId()
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\rightarrow get\_hardwareId() \\
\textit{YWakeUpMonitor} \\
wakeupmonitor\rightarrow hardwareId() \\
wakeupmonitor.get\_hardwareId()

\par Returns the unique hardware identifier of the monitor \textbf{in the form SERIAL\_FUNCTIONID.}

\par \textbf{string get\_hardwareId( )}

The unique hardware identifier is \textbf{composed of the device serial number and of the hardware identifier of the monitor (for example RELAYLO1-123456.relay1).}

\par \textbf{Returns :}
\par \hspace{1em} a string that uniquely identifies the monitor \hspace{1em} (ex: RELAYLO1-123456.relay1)

\par \textbf{On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.}
wakeupmonitor→get_logicalName()
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.
YWakeUpMonitor

wakeupmonitor→get_module()
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 online.

Returns:

an instance of YModule
3. Reference

YWakeUpMonitor

wakeupmonitor→get_nextWakeUp()
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.
Returns the maximal wake up time (in seconds) before automatically going to sleep.

<table>
<thead>
<tr>
<th>int get_powerDuration()</th>
</tr>
</thead>
</table>

**Returns:**
- an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep.

On failure, throws an exception or returns `Y_POWERDURATION_INVALID`. 
wakeupmonitor→get_sleepCountdown()
wakeupmonitor→sleepCountdown()
wakeupmonitor.get_sleepCountdown()

Returns the delay before the next sleep period.

```c
int get_sleepCountdown() {
    // Implementation
    return delay; // an integer corresponding to the delay
}
```

Returns:
- an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns Y_SLEEPCOUNTDOWN_INVALID.
### 3. Reference

`wakeupmonitor→get_userData()`

`wakeupmonitor→userData()`

`wakeupmonitor.get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

Object `getUserData()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns**:
- the object stored previously by the caller.
YWakeUpMonitor

YWakeUpMonitor

wakeupmonitor\rightarrow get\_wakeUpReason()\\
wakeupmonitor\rightarrow 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_SCHEDULE1` and `Y_WAKEUPREASON_SCHEDULE2` corresponding to the latest wake up reason.

On failure, throws an exception or returns `Y_WAKEUPREASON_INVALID`. 
3. Reference

`wakeupmonitor.get_wakeUpState()` returns the current state of the monitor.

```c
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() \[wakeupmonitor.isOnline()\] YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error.

```cpp
bool 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
3. Reference

**wakeupmonitor.load()**

Preloads the monitor cache with a specified validity duration.

**YRETCODE load( int 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.
Continues the enumeration of monitors started using \texttt{yFirstWakeUpMonitor()}.  

\texttt{YWakeUpMonitor\ nextWakeUpMonitor()} 

\textbf{Returns :}  
a pointer to a \texttt{YWakeUpMonitor} object, corresponding to a monitor currently online, or a \texttt{null} pointer if there are no more monitors to enumerate.
3. Reference

wakeupmonitor→registerValueCallback()  
wakeupmonitor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback(ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback 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() Resets the sleep countdown.

int resetSleepCountDown()

Returns:
YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.
Changes the logical name of the monitor.

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.
### 3. Reference

**YWakeUpMonitor**

**wakeupmonitor.setNextWakeUp()**  
**wakeupmonitor.setNextWakeUp()**  
**wakeupmonitor.set_nextWakeUp()**

Changes the days of the week when a wake up must take place.

```c
int set_nextWakeUp( long newval)
```

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>newval</strong>  an integer corresponding to the days of the week when a wake up must take place</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
3. Reference

**YWakeUpMonitor**

`wakeupmonitor→set_powerDuration()`  
`wakeupmonitor→setPowerDuration()`  
`wakeupmonitor.set_powerDuration()`

Changes the maximal wake up time (seconds) before automatically going to sleep.

```c
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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wakeupmonitor.set_sleepCountdown()</code></td>
<td>Changes the delay before the next sleep period.</td>
</tr>
</tbody>
</table>

```c
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.
### YWakeUpMonitor

**wakeupmonitor->set_userData()**  
**wakeupmonitor->setUserData()**  
**wakeupmonitor.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```c
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.

```c
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.
**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.

```c
int sleepFor(int secUntilWakeUp, int secBeforeSleep)
```

The count down before sleep can be canceled with `resetSleepCountDown`.

**Parameters:**
- `secUntilWakeUp` number of seconds before next wake up
- `secBeforeSleep` number of seconds before going into sleep mode

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
wakeupmonitor→sleepUntil()

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.

```c
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.
Forces a wake up.

```c
int wakeUp()
```
3.48. 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:

```javascript
<script type='text/javascript' src='yocto_wakeupschedule.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YWakeUpSchedule = yoctolib.YWakeUpSchedule;
require_once('yocto_wakeupschedule.php');
```
```cpp
#include "yocto_wakeupschedule.h"
```
```m
uses yocto_wakeupschedule;
```
```vb
yocto_wakeupschedule.vb
```
```cs
yocto_wakeupschedule.cs
```
```java
import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
```
```py
from yocto_wakeupschedule import *
```

### Global functions

- **yFindWakeUpSchedule(func)**
  - Retrieves a wake up schedule for a given identifier.

- **yFirstWakeUpSchedule()**
  - Starts the enumeration of wake up schedules currently accessible.

### YWakeUpSchedule methods

- **wakeupschedule→describe()**
  - Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **wakeupschedule→get_advertisedValue()**
  - Returns the current value of the wake up schedule (no more than 6 characters).

- **wakeupschedule→get_errorMessage()**
  - Returns the error message of the latest error with the wake up schedule.

- **wakeupschedule→get_errorType()**
  - Returns the numerical error code of the latest error with the wake up schedule.

- **wakeupschedule→get_friendlyName()**
  - Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

- **wakeupschedule→get_functionDescriptor()**
  - Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **wakeupschedule→get_functionId()**
  - Returns the hardware identifier of the wake up schedule, without reference to the module.

- **wakeupschedule→get_hardwareId()**
  - Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.

- **wakeupschedule→get_hours()**
  - Returns the hours scheduled for wake up.

- **wakeupschedule→get_logicalName()**
  - Returns the logical name of the wake up schedule.

- **wakeupschedule→get_minutes()**
  - Returns all the minutes of each hour that are scheduled for wake up.

- **wakeupschedule→get_minutesA()**
Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

```plaintext
wakeupschedule→get_minutesB()
```

Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

```plaintext
wakeupschedule→get_module()
```

Gets the YModule object for the device on which the function is located.

```plaintext
wakeupschedule→get_module_async(callback, context)
```

Gets the YModule object for the device on which the function is located (asynchronous version).

```plaintext
wakeupschedule→get_monthDays()
```

Returns the days of the month scheduled for wake up.

```plaintext
wakeupschedule→get_months()
```

Returns the months scheduled for wake up.

```plaintext
wakeupschedule→get_nextOccurence()
```

Returns the date/time (seconds) of the next wake up occurrence.

```plaintext
wakeupschedule→get_userData()
```

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```plaintext
wakeupschedule→get_weekDays()
```

Returns the days of the week scheduled for wake up.

```plaintext
wakeupschedule→isOnline()
```

Checks if the wake up schedule is currently reachable, without raising any error.

```plaintext
wakeupschedule→isOnline_async(callback, context)
```

Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

```plaintext
wakeupschedule→load(msValidity)
```

Preloads the wake up schedule cache with a specified validity duration.

```plaintext
wakeupschedule→load_async(msValidity, callback, context)
```

Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

```plaintext
wakeupschedule→nextWakeUpSchedule()
```

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

```plaintext
wakeupschedule→registerValueCallback(callback)
```

Registers the callback function that is invoked on every change of advertised value.

```plaintext
wakeupschedule→set_hours(newval)
```

Changes the hours when a wake up must take place.

```plaintext
wakeupschedule→set_logicalName(newval)
```

Changes the logical name of the wake up schedule.

```plaintext
wakeupschedule→set_minutes(bitmap)
```

Changes all the minutes where a wake up must take place.

```plaintext
wakeupschedule→set_minutesA(newval)
```

Changes the minutes in the 00-29 interval when a wake up must take place.

```plaintext
wakeupschedule→set_minutesB(newval)
```

Changes the minutes in the 30-59 interval when a wake up must take place.

```plaintext
wakeupschedule→set_monthDays(newval)
```

Changes the days of the month when a wake up must take place.

```plaintext
wakeupschedule→set_months(newval)
```

Changes the months when a wake up must take place.

```plaintext
wakeupschedule→set_userData(data)
```

Stores a user context provided as argument in the userData attribute of the function.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wakeupschedule.set_weekDays(newval)</code></td>
<td>Changes the days of the week when a wake up must take place.</td>
</tr>
<tr>
<td><code>wakeupschedule.wait_async(callback, context)</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YWakeUpSchedule.FindWakeUpSchedule()\n
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.
3. Reference

YWakeUpSchedule.FirstWakeUpSchedule()

YWakeUpSchedule.FirstWakeUpSchedule()

```
YWakeUpSchedule.FirstWakeUpSchedule()
```

Starts the enumeration of wake up schedules currently accessible.

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a pointer to a <code>YWakeUpSchedule</code> object, corresponding to the first wake up schedule currently online, or a null pointer if there are none.</td>
</tr>
</tbody>
</table>
3. Reference

`wakeschedule.describe()`  

YWakeUpSchedule

Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

`string describe( )`

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns:**

A string that describes the wake up schedule (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)**
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.
Returns the error message of the latest error with the wake up schedule.

```plaintext
Returns:
    a string corresponding to the latest error message that occurred while using the wake up schedule object
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.
ywakeschedule.get_errorType()  

YWakeUpSchedule.wakeschedule.errorType()  

wakeschedule.get_errorType()  

Returns the numerical error code of the latest error with the wake up schedule.

YRETCODE 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 example: MyCustomName.relay1)

Returns:

- a string that uniquely identifies the wake up schedule using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```c
YFUN_DESCR 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_FUNCTIONDESCRIPTION_INVALID`.
3. Reference

### \texttt{wakeupschedule}→\texttt{get\_functionId()}

\texttt{wakeupschedule}→\texttt{functionId()}

\texttt{wakeupschedule.get\_functionId()}

<table>
<thead>
<tr>
<th>YWakeUpSchedule</th>
</tr>
</thead>
</table>

Returns the hardware identifier of the wake up schedule, without reference to the module.

#### string \texttt{get\_functionId()}( )

For example \texttt{relay1}

**Returns:**

- a string that identifies the wake up schedule (ex: \texttt{relay1})

On failure, throws an exception or returns \texttt{Y\_FUNCTIONID\_INVALID}.
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 RELAY01-123456.relay1).

Returns:
- a string that uniquely identifies the wake up schedule (ex: RELAY01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
YWakeUpSchedule

Wakeupschedule\rightarrow get\_hours()

Wakeupschedule\rightarrow hours()

Wakeupschedule.get\_hours()

Returns the hours scheduled for wake up.

\begin{tabular}{|l|}
\hline
\textbf{int} \textbf{get\_hours}( ) \\
\hline
\end{tabular}

\begin{itemize}
\item \textbf{Returns :}
\begin{itemize}
\item an integer corresponding to the hours scheduled for wake up
\end{itemize}
\item On failure, throws an exception or returns Y\_HOURS\_INVALID.
\end{itemize}
3. Reference

*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.

```
3. Reference

wakeschedule→get_minutes()
wakeschedule→minutes()
wakeschedule.get_minutes()

YWakeUpSchedule

Returns all the minutes of each hour that are scheduled for wake up.

long get_minutes()
ywakeschedule\rightarrow get\_minutesA() \quad YWakeUpSchedule

ywakeschedule\rightarrow minutesA() \quad wakeupschedule.get\_minutesA()

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

int get\_minutesA() |

Returns:
an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y\_MINUTESA\_INVALID.
Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

```java
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`. 
3. Reference

YWakeUpSchedule

wakeupschedule←get_module()

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 online.

Returns:

an instance of YModule
Returns the days of the month scheduled for wake up.

```c
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`. 
YWakeUpSchedule

```
wakeupschedule->get_months()
wakeupschedule->months()
wakeupschedule.get_months()
```

Returns the months scheduled for wake up.

```c
int get_months()
```

**Returns:**

an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns Y_MONTHS_INVALID.
3. Reference

YWakeUpSchedule

wakeupschedule\rightarrow get\_nextOccurence()

wakeupschedule\rightarrow nextOccurence()

wakeupschedule.get\_nextOccurence()

Returns the date/time (seconds) of the next wake up occurrence

long get\_nextOccurence()

Returns:

an integer corresponding to the date/time (seconds) of the next wake up occurrence

On failure, throws an exception or returns Y\_NEXTOCCURRENCE\_INVALID.
`YWakeUpSchedule.get_userData()`  
`wakeupschedule.getUserData()`  
`wakeupschedule.userData()`  

Returns the value of the `userData` attribute, as previously stored using method `set_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.
Returns the days of the week scheduled for wake up.

```c
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`. 
wakeschedule→isOnline()  
wakeschedule.isOnline()

Checks if the wake up schedule is currently reachable, without raising any error.

`bool 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
Preloads the wake up schedule cache with a specified validity duration.

**YRETCODE load (int 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.
Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

**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.
YWakeUpSchedule.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback 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.
Changes the hours when a wake up must take place.

```c
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.
3. Reference

YWakeUpSchedule
wakeupschedule→set_logicalName()
wakeupschedule→setLogicalName()
wakeupschedule.set_logicalName()

Changes the logical name of the wake up schedule.

```c
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()**

Changes all the minutes where a wake up must take place.

```c
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**

Changes the minutes in the 00-29 interval when a wake up must take place.

```c
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.
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.
3. Reference

```
YWakeUpSchedule
wakeupschedule.set_monthDays()
wakeupschedule.setMonthDays()
```

Changes the days of the month when a wake up must take place.

```c
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.
Changes the months when a wake up must take place.

```c
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.
Stores a user context provided as argument in the userData attribute of the function.

```java
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
YWakeUpSchedule
wakeupschedule→set_weekDays()
wakeupschedule→setWeekDays()
wakeupschedule.set_weekDays()

Changes the days of the week when a wake up must take place.

<table>
<thead>
<tr>
<th>int set_weekDays( int newval)</th>
</tr>
</thead>
</table>

**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. Reference

3.49. 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:

<table>
<thead>
<tr>
<th>js</th>
<th><code>&lt;script type='text/javascript' src='yocto_watchdog.js'&gt;&lt;/script&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>nodejs</td>
<td><code>var yoctolib = require('yoctolib');</code></td>
</tr>
<tr>
<td>php</td>
<td><code>require_once('yocto_watchdog.php');</code></td>
</tr>
<tr>
<td>cpp</td>
<td><code>#include &quot;yocto_watchdog.h&quot;</code></td>
</tr>
<tr>
<td>m</td>
<td><code>#import &quot;yocto_watchdog.h&quot;</code></td>
</tr>
<tr>
<td>pas</td>
<td><code>uses yocto_watchdog;</code></td>
</tr>
<tr>
<td>vb</td>
<td><code>yocto_watchdog.vb</code></td>
</tr>
<tr>
<td>cs</td>
<td><code>yocto_watchdog.cs</code></td>
</tr>
<tr>
<td>java</td>
<td><code>import com.yoctopuce.YoctoAPI.YWatchdog;</code></td>
</tr>
<tr>
<td>py</td>
<td><code>from yocto_watchdog import *</code></td>
</tr>
</tbody>
</table>

### Global functions

<table>
<thead>
<tr>
<th><code>yFindWatchdog(func)</code></th>
<th>Retrieves a watchdog for a given identifier.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFirstWatchdog()</code></td>
<td>Starts the enumeration of watchdog currently accessible.</td>
</tr>
</tbody>
</table>

### YWatchdog methods

<table>
<thead>
<tr>
<th><code>watchdog→delayedPulse(ms_delay, ms_duration)</code></th>
<th>Schedules a pulse.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>watchdog→describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the watchdog in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>watchdog→get_advertisedValue()</code></td>
<td>Returns the current value of the watchdog (no more than 6 characters).</td>
</tr>
<tr>
<td><code>watchdog→get_autoStart()</code></td>
<td>Returns the watchdog running state at module power on.</td>
</tr>
<tr>
<td><code>watchdog→get_countdown()</code></td>
<td>Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.</td>
</tr>
<tr>
<td><code>watchdog→get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the watchdog.</td>
</tr>
<tr>
<td><code>watchdog→get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the watchdog.</td>
</tr>
<tr>
<td><code>watchdog→get_friendlyName()</code></td>
<td>Returns a global identifier of the watchdog in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><code>watchdog→get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><code>watchdog→get_functionId()</code></td>
<td>Returns the hardware identifier of the watchdog, without reference to the module.</td>
</tr>
</tbody>
</table>
**3. Reference**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>watchdog-&gt;get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the watchdog.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_maxTimeOnStateA()</code></td>
<td>Returns the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_maxTimeOnStateB()</code></td>
<td>Returns the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_module_async(callback, context)</code></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_output()</code></td>
<td>Returns the output state of the watchdog, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_pulseTimer()</code></td>
<td>Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_running()</code></td>
<td>Returns the watchdog running state.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_state()</code></td>
<td>Returns the state of the watchdog (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_stateAtPowerOn()</code></td>
<td>Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_triggerDelay()</code></td>
<td>Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_triggerDuration()</code></td>
<td>Returns the duration of resets caused by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td><code>watchdog-&gt;get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><code>watchdog-&gt;isOnline()</code></td>
<td>Checks if the watchdog is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>watchdog-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the watchdog is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>watchdog-&gt;load(msValidity)</code></td>
<td>Preloads the watchdog cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>watchdog-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the watchdog cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>watchdog-&gt;nextWatchdog()</code></td>
<td>Continues the enumeration of watchdog started using yFirstWatchdog().</td>
</tr>
<tr>
<td><code>watchdog-&gt;pulse(ms_duration)</code></td>
<td>Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).</td>
</tr>
<tr>
<td><code>watchdog-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>Function Call</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>watchdog.resetWatchdog()</td>
<td>Resets the watchdog.</td>
</tr>
<tr>
<td>watchdog.set_autoStart(newval)</td>
<td>Changes the watchdog running state at module power on.</td>
</tr>
<tr>
<td>watchdog.set_logicalName(newval)</td>
<td>Changes the logical name of the watchdog.</td>
</tr>
<tr>
<td>watchdog.set_maxTimeOnStateA(newval)</td>
<td>Sets the maximum time (ms) allowed for THEFUNCTIONS to stay in state A before automatically switching back in to B state.</td>
</tr>
<tr>
<td>watchdog.set_maxTimeOnStateB(newval)</td>
<td>Sets the maximum time (ms) allowed for THEFUNCTIONS to stay in state B before automatically switching back in to A state.</td>
</tr>
<tr>
<td>watchdog.set_output(newval)</td>
<td>Changes the output state of the watchdog, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td>watchdog.set_running(newval)</td>
<td>Changes the running state of the watchdog.</td>
</tr>
<tr>
<td>watchdog.set_state(newval)</td>
<td>Changes the state of the watchdog (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td>watchdog.set_stateAtPowerOn(newval)</td>
<td>Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).</td>
</tr>
<tr>
<td>watchdog.set_triggerDelay(newval)</td>
<td>Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td>watchdog.set_triggerDuration(newval)</td>
<td>Changes the duration of resets caused by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td>watchdog.set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>watchdog.wait_async(callback, context)</td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</td>
</tr>
</tbody>
</table>
YWatchdog.FindWatchdog()

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()

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()**

Schedules a pulse.

```c
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.
YWatchdog

Returns a short text that describes unambiguously the instance of the watchdog in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
string describe()
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module 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)
Returns the current value of the watchdog (no more than 6 characters).

```java
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`. 
YWatchdog

watchdog\rightarrow get\_autoStart()

\text{watchdog\rightarrow autoStart()} \text{watchdog.get\_autoStart()}

Returns the watchdog running state at module power on.

\begin{verbatim}
int get_autoStart( )
\end{verbatim}

\textbf{Returns}:

- either \texttt{Y\_AUTOSTART\_OFF} or \texttt{Y\_AUTOSTART\_ON}, according to the watchdog running state at module power on

On failure, throws an exception or returns \texttt{Y\_AUTOSTART\_INVALID}.
Returns the number of milliseconds remaining before a pulse (delayedPulse() call). When there is no scheduled pulse, returns zero.

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.
### 3. Reference

```plaintext
YWatchdog

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_errorMessage()</code></td>
</tr>
<tr>
<td><code>errorMessage()</code></td>
</tr>
<tr>
<td><code>get_errorMessage()</code></td>
</tr>
</tbody>
</table>

**YWatchdog**

Returns the error message of the latest error with the watchdog.

```plaintext
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
3. Reference

YWatchdog

\texttt{watchdog.get\_errorType()}

\texttt{YWatchdog.errorType()}

\texttt{watchdog.get\_errorType()}

Returns the numerical error code of the latest error with the watchdog.

\texttt{YRETCODE get\_errorType()}

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

\textbf{Returns :}

a number corresponding to the code of the latest error that occurred while using the watchdog object
3. Reference

YWatchdog
watchdog→get_friendlyName()
watchdog→friendlyName()
watchdog.get_friendlyName()

Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.

```java
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`.
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR 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.
Returns the hardware identifier of the watchdog, without reference to the module.

For example relay1

**Returns:**

- A string that identifies the watchdog (ex: relay1)

On failure, throws an exception or returns \texttt{Y\_FUNCTIONID\_INVALID}.
3. Reference

**YWatchdog**

```markdown
watchdog->get_hardwareId()
watchdog->hardwareId() watchdog.get_hardwareId()
```

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

```markdown
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`.
**get_logicalName()**

Returns the logical name of the watchdog.

```csharp
string get_logicalName()
```

**Returns:**
- a string corresponding to the logical name of the watchdog.

On failure, throws an exception or returns **Y_LOGICALNAME_INVALID**.
3. Reference

YWatchdog

let watchdog→get_maxTimeOnStateA()

let watchdog→maxTimeOnStateA()

let 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.
Retourne the maximum time (ms) allowed for $\text{THEFUNCTIONS}$ 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 $\text{Y_MAXTIMEONSTATEB_INVALID}$. 

**3. Reference**

$\text{YWatchdog}$

$\text{watchdog} \rightarrow \text{get\_maxTimeOnStateB()}$

$\text{watchdog} \rightarrow \text{maxTimeOnStateB()}$

$\text{watchdog}.\text{get\_maxTimeOnStateB()}$
Gets the `YModule` object for the device on which the function is located.

**YModule get_module( )**

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns:**

an instance of `YModule`
Returns the output state of the watchdog, when used as a simple switch (single throw).

```c
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`.
Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

<table>
<thead>
<tr>
<th>long get_pulseTimer( )</th>
</tr>
</thead>
</table>

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**.
YWatchdog

**get_running()**

Returns the watchdog running state.

```c
int get_running()
```

**running()**

** watchdog.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`. 
Returns the state of the watchdog (A for the idle position, B for the active position).

```c
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`.
3. Reference

YWatchdog

watchdog→get_stateAtPowerOn()
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).

```c
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`. 
**>Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

```c
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`. 
**YWatchdog**

**watchdog → get_triggerDuration()**

**watchdog → triggerDuration()**

**watchdog.get_triggerDuration()**

Returns the duration of resets caused by the watchdog, in milliseconds.

```java
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`. 
Returns the value of the userData attribute, as previously stored using method `set_userData`.

```python
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.
YWatchdog

/*
* Checks if the watchdog is currently reachable, without raising any error.
* 
* `bool 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()

Preloads the watchdog cache with a specified validity duration.

YRETCODE load( int 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.
3. Reference

**YWatchdog nextWatchdog()**

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

**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.
Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

**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.
YWatchdog

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback 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.
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.
Changes the watchdog running state at module power on.

```c
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 running state at module power on.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the logical name of the watchdog.

```c
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.
### Reference

**YWatchdog**

**watchdog→set_maxTimeOnStateA()**

**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.

```c
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.
Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```c
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.
Changes the output state of the watchdog, when used as a simple switch (single throw).

```c
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.
Changes the running state of the watchdog.

```c
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.
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.
Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```c
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.
3. Reference

**YWatchdog**

### set_triggerDelay()

- **setTriggerDelay()**
- **set_triggerDelay()**

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

```c
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.
Changes the duration of resets caused by the watchdog, in milliseconds.

```c
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.
3. Reference

YWatchdog

watchdog→set_userData()
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
3.50. Wireless function interface

YWWireless 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:

```javascript
<script type='text/javascript' src='yocto_wireless.js'></script>
```

```nodejs
var yoctolib = require('yoctolib');
var YWireless = yoctolib.YWireless;
require_once('yocto_wireless.php');
```

```cpp
#include "yocto_wireless.h"
```

```m
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.
3. Reference

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→softAPNetwork(ssid, securityKey)
Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

wireless→wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
Retrieves a wireless lan interface for a given identifier.

**YWireless FindWireless**

```go
func FindWireless(func string) YWireless
```

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.
3. Reference

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()

YWireless.FirstWireless()
wireless → adhocNetwork() → wireless.adhocNetwork()

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

```c
int adhocNetwork( string ssid, string securityKey)
```

On the YoctoHub-Wireless-g, it is best to use softAPNetworkInstead(), which emulates an access point (Soft AP) which is more efficient and more widely supported than ad-hoc networks.

When a security key is specified for an ad-hoc network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters:**
- **ssid** the name of the network to connect to
- **securityKey** the network key, as a character string

**Returns:**
- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
wireless.describe() Returns a short text that describes unambiguously the instance of the wireless lan interface in the form TYPE(NAME)=SERIAL.FUNCTIONID.

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 wireless lan interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
wireless→get_advertisedValue()
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.
YWireless

wireless→get_channel()
wireless→channel() wireless.get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

```c
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`. 
Returns a list of YWlanRecord objects that describe detected Wireless networks.

### List<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()  
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()
wireless → errorType()
wireless.get_errorType()  

Returns the numerical error code of the latest error with the wireless lan interface.

YRETCODE 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
### 3. Reference

<table>
<thead>
<tr>
<th>wireless→get_friendlyName()</th>
<th>YWirelesswireless→friendlyName()wireless.get_friendlyName()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Returns</strong></td>
<td>a global identifier of the wireless lan interface in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><strong>string get_friendlyName( )</strong></td>
<td>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)</td>
</tr>
<tr>
<td><strong>Returns :</strong></td>
<td>a string that uniquely identifies the wireless lan interface using logical names (ex: MyCustomName.relay1)</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.</td>
</tr>
</tbody>
</table>
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

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`. 
3. Reference

wireless \rightarrow \text{get\_functionId()} \quad \text{YWireless}
wireless \rightarrow \text{functionId()} \quad \text{wireless.get\_functionId()}

Returns the hardware identifier of the wireless lan interface, without reference to the module.

\begin{verbatim}
string get_functionId()
\end{verbatim}

For example \texttt{relay1}

\begin{tabular}{|p{0.9\textwidth}|}
\hline
\textbf{Returns} : \\
\hspace{1cm} a string that identifies the wireless lan interface (ex: \texttt{relay1})\\
\hline
\end{tabular}

On failure, throws an exception or returns \texttt{Y\_FUNCTIONID\_INVALID}.
3. Reference

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.
 Returns the link quality, expressed in percent.

\[
\text{int } \text{get\_link\_Quality}( )
\]

**Returns:**
- an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns Y\_LINKQUALITY\_INVALID.
wireless \rightarrow \text{get\_logical\_Name()}
wireless\rightarrow \text{logical\_Name()}
wireless.get\_logical\_Name()

Returns the logical name of the wireless lan interface.

\text{string get\_logical\_Name( )}

\textbf{Returns :}

a string corresponding to the logical name of the wireless lan interface.

On failure, throws an exception or returns \texttt{Y\_LOGICAL\_NAME\_INVALID}.
Returns the latest status message from the wireless interface.

```c
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
wireless
wireless
wireless

wireless->get_module()
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 online.

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 \( \rightarrow \) get_ssid()
wireless \( \rightarrow \) ssid() wireless.get_ssid()

Returns the wireless network name (SSID).

string \textbf{get_ssid}() \\

\textbf{Returns}:
\begin{itemize}
  \item a string corresponding to the wireless network name (SSID)
\end{itemize}

On failure, throws an exception or returns \texttt{YSSID_INVALID}.
Returns the value of the userData attribute, as previously stored using method `set_userData`.

```javascript
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()

Checks if the wireless lan interface is currently reachable, without raising any error.

```c
bool 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()

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

```cpp
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` when the call succeeds.

On failure, throws an exception or returns a negative error code.
wireless->load() \( \text{wireless.load()} \)

Preloads the wireless lan interface cache with a specified validity duration.

\[
\text{YRETCODE load( int msValidity)}
\]

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.
<table>
<thead>
<tr>
<th>YWireless → nextWireless()</th>
<th>YWireless</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continues the enumeration of wireless lan interfaces started using yFirstWireless().</td>
<td></td>
</tr>
</tbody>
</table>

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()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( ValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback 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.
### Changes the logical name of the wireless lan interface.

```c
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()`
wireless → `setUserData()`
wireless.setUserData()

Stores a user context provided as argument in the `userData` attribute of the function.

```c
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
Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

```c
int softAPNetwork( string ssid, string securityKey)
```

This function can only be used with the YoctoHub-Wireless-g.

When a security key is specified for a SoftAP network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters:**
- `ssid` the name of the network to connect to
- `securityKey` the network key, as a character string

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
Index

A
Accelerometer 33
adhocNetwork, YWireless 1756
Altitude 75
AnButton 117

B
Blueprint 12
brakingForceMove, YMotor 869

C
C# 3
calibrate, YLightSensor 735
calibrateFromPoints, YAccelerometer 37
calibrateFromPoints, YAltitude 79
calibrateFromPoints, YCarbonDioxide 159
calibrateFromPoints, YCompass 227
calibrateFromPoints, YCurrent 267
calibrateFromPoints, YGenericSensor 544
calibrateFromPoints, YGyro 593
calibrateFromPoints, YHumidity 669
calibrateFromPoints, YLightSensor 736
calibrateFromPoints, YMagnetometer 777
calibrateFromPoints, YPower 990
calibrateFromPoints, YPressure 1033
calibrateFromPoints, YPwmInput 1072
calibrateFromPoints, YQt 1181
calibrateFromPoints, YSensor 1319
calibrateFromPoints, YTilt 1491
calibrateFromPoints, YVoc 1530
calibrateFromPoints, YVoltage 1569
callbackLogin, YNetwork 911
cancel3DCalibration, YRefFrame 1247
CarbonDioxide 155
checkFirmware, YModule 825
CheckLogicalName, YAPI 14
clear, YDisplayLayer 457
clearConsole, YDisplayLayer 458
Clock 1216
ColorLed 194
Compass 223
Configuration 1243
consoleOut, YDisplayLayer 459
copyLayerContent, YDisplay 413
Current 263
delayedPulse, YWatchdog 1712
describe, YAccelerometer 38
describe, YAltitude 80
describe, YAnButton 121
describe, YCarbonDioxide 160
describe, YColorLed 197
describe, YCompass 228
describe, YCurrent 268
describe, YDataLogger 306
describe, YDigitalIO 370
describe, YDisplay 414
describe, YDualPower 491
describe, YFiles 516
describe, YGenericSensor 545
describe, YGyro 594
describe, YHubPort 643
describe, YHumidity 670
describe, YLed 707
describe, YLightSensor 737
describe, YMagnetometer 778
describe, YModule 826
describe, YMotor 870
describe, YNetwork 912
describe, YOsControl 966
describe, YPower 991
describe, YPressure 1034
describe, YPwmInput 1073
describe, YPwmOutput 1120
describe, YPwmPowerSource 1157
describe, YQt 1182
describe, YRealTimeClock 1219
describe, YRefFrame 1248
describe, YRelay 1284
describe, YSensor 1320
describe, YSerialPort 1359
describe, YServo 1415
describe, YTemperature 1451
describe, YTilt 1492
describe, YVoc 1531
describe, YVoltage 1570
describe, YVSource 1607
describe, YWakeUpMonitor 1640
describe, YWakeUpSchedule 1675
describe, YWatchdog 1713
describe, YWireless 1757
Digital 365
DisableExceptions, YAPI 15
Display 409
DisplayLayer 456
drawBar, YDisplayLayer 460
drawBitmap, YDisplayLayer 461
drawCircle, YDisplayLayer 462
drawDisc, YDisplayLayer 463
drawImage, YDisplayLayer 464
drawPixel, YDisplayLayer 465

D
Data 336, 338, 350
DataLogger 302
delayedPulse, YDigitalIO 369
delayedPulse, YRelay 1283
drawRect, YDisplayLayer 466
drawText, YDisplayLayer 467
drivingForceMove, YMotor 871
dutyCycleMove, YPwmOutput 1121

E
EnableExceptions, YAPI 16
Error 8
External 488

F
fade, YDisplay 415
Files 513
FindAccelerometer, YAccelerometer 35
FindAltitude, YAltitude 77
FindAnButton, YAnButton 119
FindCarbonDioxide, YCarbonDioxide 157
FindColorLed, YColorLed 195
FindCompass, YCompass 225
FindCurrent, YCurrent 265
FindDataLogger, YDataLogger 304
FindDigitalIO, YDigitalIO 367
FindDisplay, YDisplay 411
FindDualPower, YDualPower 490
FindFiles, YFiles 515
FindGenericSensor, YGenericSensor 542
FindGyro, YGyro 591
FindHubPort, YHubPort 641
FindHumidity, YHumidity 667
FindLed, YLed 705
FindLightSensor, YLightSensor 734
FindMagnetometer, YMagnetometer 776
FindModule, YModule 824
FindMotor, YMotor 868
FindNetwork, YNetwork 910
FindOsControl, YOsControl 965
FindPower, YPower 989
FindPressure, YPressure 1032
FindPwmInput, YPwmInput 1071
FindPwmOutput, YPwmOutput 1119
FindPwmPowerSource, YPwmPowerSource 1156
FindQt, YQt 1180
FindRealTimeClock, YRealTimeClock 1218
FindRefFrame, YRefFrame 1246
FindRelay, YRelay 1282
FindSensor, YSensor 1318
FindSerialPort, YSerialPort 1358
FindServo, YServo 1414
FirstAccelerometer, YAccelerometer 36
FirstAltitude, YAltitude 78
FirstAnButton, YAnButton 120
FirstCarbonDioxide, YCarbonDioxide 158
FirstColorLed, YColorLed 196
FirstCompass, YCompass 226
FirstCurrent, YCurrent 266
FirstDataLogger, YDataLogger 305
FirstDigitalIO, YDigitalIO 368
FirstDisplay, YDisplay 412
FirstDualPower, YDualPower 490
FirstFiles, YFiles 515
FirstGenericSensor, YGenericSensor 543
FirstGyro, YGyro 592
FirstHubPort, YHubPort 642
FirstHumidity, YHumidity 668
FirstLed, YLed 706
FirstLightSensor, YLightSensor 734
FirstMagnetometer, YMagnetometer 776
FirstModule, YModule 824
FirstMotor, YMotor 868
FirstNetwork, YNetwork 910
FirstOsControl, YOsControl 965
FirstPower, YPower 989
FirstPressure, YPressure 1032
FirstPwmInput, YPwmInput 1071
FirstPwmOutput, YPwmOutput 1119
FirstPwmPowerSource, YPwmPowerSource 1156
FirstQt, YQt 1180
FirstRealTimeClock, YRealTimeClock 1218
FirstRefFrame, YRefFrame 1246
FirstRelay, YRelay 1282
FirstSensor, YSensor 1318
FirstSerialPort, YSerialPort 1358
FirstServo, YServo 1414
FirstTemperature, YTemperature 1449
FirstTilt, YTilt 1490
FirstVoc, YVoc 1529
FirstVoltage, YVoltage 1568
FirstVSource, YVSource 1606
FirstWakeUpMonitor, YWakeUpMonitor 1639
FirstWakeUpSchedule, YWakeUpSchedule 1674
FirstWatchdog, YWatchdog 1711
FirstWireless, YWireless 1755
forgetAllDataStreams, YDataLogger 307
format_fs, YFiles 517
Formatted 336
Frame 1243
FreeAPI, YAPI 17
functionCount, YModule 827
functionId, YModule 828
functionName, YModule 829
Functions 13
functionValue, YModule 830

G
General 13
GenericSensor 540
get_functionId, YMagnetometer 786
get_functionId, YMotor 882
get_functionId, YNetwork 926
get_functionId, YOsControl 972
get_functionId, YPower 1000
get_functionId, YPressure 1042
get_functionId, YPwmInput 1083
get_functionId, YPwmOutput 1132
get_functionId, YPwmPowerSource 1164
get_functionId, YQt 1190
get_functionId, YRealTimeClock 1226
get_functionId, YRefFrame 1260
get_functionId, YRelay 1291
get_functionId, YSensor 1328
get_functionId, YSerialPort 1367
get_functionId, YPower 1423
get_functionId, YTemperature 1459
get_functionId, YTilt 1500
get_functionId, YVoltage 1578
get_functionId, YVSource 1615
get_functionId, YMagnetometer 1746
get_functionId, YMotor 1793
get_functionId, YNetwork 1837
get_functionId, YOsControl 1882
get_functionId, YPwmInput 1924
get_functionId, YPwmOutput 1975
get_functionId, YPwmPowerSource 2006
get_functionId, YEg 2037
get_functionId, YVoc 2078
get_functionId, YVoltage 2119
get_functionId, YVSource 2160
get_functionId, YWakeUpMonitor 2191
get_functionId, YWakeUpSchedule 2232
get_functionId, YWatchdog 2273
get_functionId, YWireless 2314
get_hardwareId, YAccelerometer 48
get_hardwareId, YAltitude 90
get_hardwareId, YAnButton 132
get_hardwareId, YCarbonDioxide 170
get_hardwareId, YColorLed 204
get_hardwareId, YCompass 238
get_hardwareId, YDataLogger 319
get_hardwareId, YDataSet 341
get_hardwareId, YDigitalIO 381
get_hardwareId, YDisplay 428
get_hardwareId, YDisplayLayer 471
get_hardwareId, YDualPower 499
get_hardwareId, YFiles 526
get_hardwareId, YGenericSensor 554
get_hardwareId, YGyro 603
get_hardwareId, YHubPort 652
get_hardwareId, YHumidity 679
get_hardwareId, YLed 715
get_hardwareId, YLightSensor 746
get_hardwareId, YMagnetometer 787
get_hardwareId, YModule 835
get_hardwareId, YMotor 883
get_hardwareId, YNetwork 927
get_hardwareId, YOsControl 973
get_hardwareId, YPower 1001
get_hardwareId, YPressure 1042
get_hardwareId, YPwmInput 1083
get_hardwareId, YPwmOutput 1132
get_hardwareId, YPwmPowerSource 1164
get_hardwareId, YQt 1190
get_hardwareId, YRealTimeClock 1227
get_hardwareId, YRefFrame 1261
get_hardwareId, YRelay 1292
get_hardwareId, YSensor 1329
get_hardwareId, YSerialPort 1368
get_hardwareId, YServo 1424
get_hardwareId, YTemperature 1460
get_hardwareId, YTilt 1501
get_hardwareId, YVoc 1540
get_hardwareId, YVoltage 1579
get_hardwareId, YVSource 1616
get_hardwareId, YWakeUpMonitor 1647
get_hardwareId, YWakeUpSchedule 1682
get_hardwareId, YWatchdog 1722
get_hardwareId, YWireless 1766
get_heading, YGyro 604
get_highestValue, YAccelerometer 48
get_highestValue, YAltitude 90
get_highestValue, YCarbonDioxide 170
get_highestValue, YCompass 238
get_highestValue, YCurrent 278
get_highestValue, YGenericSensor 555
get_highestValue, YGyro 605
get_highestValue, YHumidity 680
get_highestValue, YLightSensor 747
get_highestValue, YMagnetometer 788
get_highestValue, YPower 1002
get_highestValue, YPressure 1044
get_highestValue, YPwmInput 1085
get_highestValue, YQt 1192
get_highestValue, YSensor 1330
get_highestValue, YTilt 1502
get_highestValue, YVoltage 1541
get_highestValue, YVoltage 1580
get_hours, YWakeUpSchedule 1683
get_hslColor, YColorLed 205
get_ipAddress, YNetwork 928
get_isPressed, YAnButton 133
get_lastLogs, YModule 836
get_lastMsg, YSerialPort 1369
get_lastTimePressed, YAnButton 134
get_lastTimeReleased, YAnButton 135
get_layerCount, YDisplay 429
get_layerHeight, YDisplay 430
get_layerHeight, YDisplayLayer 471
get_layerWidth, YDisplay 431
get_layerWidth, YDisplayLayer 472
get_linkQuality, YWireless 1767
get_list, YFiles 527
get_logFrequency, YAccelerometer 49
get_logFrequency, YAltitude 91
get_logFrequency, YCarbonDioxide 171
get_logFrequency, YCompass 239
get_logFrequency, YCurrent 279
get_logFrequency, YGenericSensor 556
get_logFrequency, YGyro 606
get_logFrequency, YHumidity 681
get_logFrequency, YLightSensor 748
get_logFrequency, YMagnetometer 789
get_logFrequency, YPower 1003
get_logFrequency, YPressure 1045
get_logFrequency, YPwmInput 1086
get_userData, YWakeUpMonitor 1653
get_userData, YWakeUpSchedule 1692
get_userData, YWatchdog 1734
get_userData, YWireless 1773
get_userPassword, YNetwork 939
get_userVar, YModule 848
get_utcOffset, YRealTimeClock 1233
get_valueRange, YGenericSensor 569
get_voltage, YVSource 1625
get_wakeUpReason, YWakeUpMonitor 1654
get_wakeUpState, YWakeUpMonitor 1655
get_weekDays, YWakeUpSchedule 1693
get_wwwWatchdogDelay, YNetwork 940
get_xValue, YAccelerometer 58
get_xValue, YGyro 621
get_xValue, YMagnetometer 798
get_yValue, YAccelerometer 59
get_yValue, YGyro 622
get_yValue, YMagnetometer 799
get_zValue, YAccelerometer 60
get_zValue, YGyro 623
get_zValue, YMagnetometer 800
GetAPIVersion, YAPI 18
GetTickCount, YAPI 19
Gyroscope 589

H
HandleEvents, YAPI 20
hide, YDisplayLayer 473
hslMove, YColorLed 211
Humidity 665

I
InitAPI, YAPI 21
Installation 3
Introduction 1
isOnline, YAccelerometer 61
isOnline, YAltitude 101
isOnline, YAnButton 143
isOnline, YCarbonDioxide 180
isOnline, YColorLed 212
isOnline, YCompass 249
isOnline, YCurrent 288
isOnline, YDataLogger 325
isOnline, YDigitalIO 391
isOnline, YDisplay 437
isOnline, YDualPower 505
isOnline, YFiles 531
isOnline, YGenericSensor 570
isOnline, YGyro 624
isOnline, YHubPort 657
isOnline, YHumidity 690

J
joinNetwork, YWireless 1775

K
keepAlive, YMotor 891

L
LightSensor 731
lineTo, YDisplayLayer 474
load, YAccelerometer 62
load, YAltitude 102
load, YAnButton 144
load, YCarbonDioxide 181
load, YColorLed 213
load, YCompass 250
load, YCurrent 289
load, YDataLogger 326
load, YDigitalIO 392
load, YDisplay 438
load, YDualPower 506
load, YFiles 532
load, YGenericSensor 571
load, YGyro 625
load, YHubPort 658
load, YHumidity 691
load, YLed 722
load, YLightSensor 759
load, YMagnetometer 802
load, YModule 850
load, YMotor 892
load, YNetwork 942
load, YOsControl 979
load, YPower 1015
load, YPressure 1055
load, YPwmInput 1101
load, YPwmOutput 1140
load, YPwmPowerSource 1170
load, YQt 1203
load, YRealTimeClock 1235
load, YRefFrame 1268
load, YRelay 1303
load, YSensor 1341
load, YSerialPort 1379
load, YServo 1433
load, YTemperature 1473
load, YTilt 1513
load, YVoltage 1591
load, YVSourse 1627
load, YWakeUpMonitor 1657
load, YWakeUpSchedule 1695
load, YWatchdog 1736
load, YWireless 1776
loadCalibrationPoints, YAccelerometer 63
loadCalibrationPoints, YAltitude 103
loadCalibrationPoints, YCarbonDioxide 182
loadCalibrationPoints, YCompass 251
loadCalibrationPoints, YCurrent 290
loadCalibrationPoints, YGyro 626
loadCalibrationPoints, YHumidity 692
loadCalibrationPoints, YLightSensor 760
loadCalibrationPoints, YMagnetometer 803
loadCalibrationPoints, YPower 1016
loadCalibrationPoints, YPressure 1056
loadCalibrationPoints, YPwmInput 1102
loadCalibrationPoints, YQt 1204
loadCalibrationPoints, YSensor 1342
loadCalibrationPoints, YTemperature 1474
loadCalibrationPoints, YTilt 1514
loadCalibrationPoints, YVoc 1553
loadCalibrationPoints, YVoltage 1592
loadMore, YDataSet 348

M
Magnetometer 773
Measured 815
modbusReadBits, YSerialPort 1380
modbusReadInputBits, YSerialPort 1381
modbusReadInputRegisters, YSerialPort 1382
modbusReadRegisters, YSerialPort 1383
modbusWriteAndReadRegisters, YSerialPort 1384
modbusWriteBit, YSerialPort 1385
modbusWriteBits, YSerialPort 1386
modbusWriteRegister, YSerialPort 1387
modbusWriteRegisters, YSerialPort 1388

N
Network 906
newSequence, YDisplay 439
nextAccelerometer, YAccelerometer 64
nextAltitude, YAltitude 104
nextAnButton, YAnButton 145
nextCarbonDioxide, YCarbonDioxide 183
nextColorLed, YColorLed 214
nextCompass, YCompass 252
nextCurrent, YCurrent 291
nextDataLogger, YDataLogger 327
nextDigitalIO, YDigitalIO 393
nextDisplay, YDisplay 440
nextDualPower, YDualPower 507
nextFiles, YFiles 533
nextGenericSensor, YGenericSensor 573
nextGyro, YGyro 627
nextHubPort, YHubPort 659
nextHumidity, YHumidity 693
nextLed, YLed 723
nextLightSensor, YLightSensor 761
nextMagnetometer, YMagnetometer 804
nextModule, YModule 851
nextMotor, YMotor 893
nextNetwork, YNetwork 943
nextOsControl, YOsControl 980
nextPower, YPower 1017
nextPressure, YPressure 1057
nextPwmInput, YPwmInput 1103
nextPwmOutput, YPwmOutput 1141
nextPwmPowerSource, YPwmPowerSource 1171
nextQt, YQt 1205
nextRealTimeClock, YRealTimeClock 1236
nextRefFrame, YRefFrame 1270
nextRelay, YRelay 1304
nextSensor, YSensor 1343
nextSerialPort, YSerialPort 1389
nextServo, YServo 1435
nextTemperature, YTemperature 1475
nextTilt, YTilt 1515
nextVoc, YVoc 1554
nextVoltage, YVoltage 1593
nextVSource, YVSource 1628
nextWakeUpMonitor, YWakeUpMonitor 1658
nextWakeUpSchedule, YWakeUpSchedule 1696
nextWatchdog, YWatchdog 1737
nextWireless, YWireless 1777

O
Object 456
P
pauseSequence, YDisplay 441
ping, YNetwork 944
playSequence, YDisplay 442
Port 640
Power 488, 986
PreregisterHub, YAPI 22
Pressure 1029
Project 3
pulse, YDigitalIO 394
pulse, YRelay 1305
pulse, YWatchdog 1738
pulseDurationMove, YPwmOutput 1142
PwmInput 1068
PwmPowerSource 1154

Q
Quaternion 1177
queryLine, YSerialPort 1390
queryMODBUS, YSerialPort 1391

R
read_seek, YSerialPort 1396
readHex, YSerialPort 1392
readLine, YSerialPort 1393
readMessages, YSerialPort 1394
readStr, YSerialPort 1395
Real 1216
reboot, YModule 852
Recorded 338
Reference 12, 1243
registerAnglesCallback, YGyro 628
RegisterDeviceArrivalCallback, YAPI 23
RegisterDeviceRemovalCallback, YAPI 24
RegisterHub, YAPI 25
RegisterHubDiscoveryCallback, YAPI 26
registerLogCallback, YModule 853
RegisterLogFunction, YAPI 27
registerQuaternionCallback, YGyro 629
registerTimedReportCallback, YAccelerometer 65
registerTimedReportCallback, YAltitude 105
registerTimedReportCallback, YCarbonDioxide 184
registerTimedReportCallback, YCompass 253
registerTimedReportCallback, YCurrent 292
registerTimedReportCallback, YGenericSensor 574
registerTimedReportCallback, YGyro 630
registerTimedReportCallback, YHumidity 694
registerTimedReportCallback, YLightSensor 762
registerTimedReportCallback, YMagnetometer 805
registerTimedReportCallback, YPower 1018
registerTimedReportCallback, YPressure 1058
registerTimedReportCallback, YPwmInput 1104
registerTimedReportCallback, YQt 1206
registerTimedReportCallback, YSensor 1344
registerTimedReportCallback, YTemperature 1476
registerTimedReportCallback, YTilt 1516
registerTimedReportCallback, YVoc 1555
registerTimedReportCallback, YVoltage 1594
registerValueCallback, YAccelerometer 66
registerValueCallback, YAltitude 106
registerValueCallback, YAnButton 146
registerValueCallback, YCarbonDioxide 185
registerValueCallback, YColorLed 215
registerValueCallback, YCurrent 293
registerValueCallback, YDataLogger 328
registerValueCallback, YDigitalIO 395
registerValueCallback, YDisplay 443
registerValueCallback, YDualPower 508
registerValueCallback, YFiles 534
registerValueCallback, YGenericSensor 575
registerValueCallback, YGyro 631
registerValueCallback, YHubPort 660
registerValueCallback, YHumidity 695
registerValueCallback, YLed 724
registerValueCallback, YLightSensor 763
registerValueCallback, YMagnetometer 806
registerValueCallback, YMotor 894
registerValueCallback, YNetwork 945
registerValueCallback, YOsControl 981
registerValueCallback, YPressure 1059
registerValueCallback, YPower 1019
registerValueCallback, YReadSeek 1339
registerValueCallback, YRealTimeClock 1237
registerValueCallback, YRefFrame 1271
registerValueCallback, YSerialPort 1397
registerValueCallback, YSensor 1345
registerValueCallback, YSerialPort 1397
registerValueCallback, YSerialPort 1397
registerValueCallback, YServo 1436
registerValueCallback, YTemperature 1477
registerValueCallback, YTime 1517
registerValueCallback, YVoc 1556
registerValueCallback, YVoltage 1595
registerValueCallback, YVSource 1630
registerValueCallback, YWakeUpMonitor 1659
registerValueCallback, YWakeUpSchedule 1697
registerValueCallback, YWatchdog 1739
registerValueCallback, YWireless 1778
Relay 1279
remove, YFiles 535
reset, YDisplayLayer 476
reset, YPower 1020
reset, YSerialPort 1398
resetAll, YDisplay 444
resetCounter, YAnButton 147
resetCounter, YPwmInput 1106
resetSleepCountDown, YWakeUpMonitor 1660
resetStatus, YMotor 895
resetWatchdog, YWatchdog 1740
revertFromFlash, YModule 854
rgbMove, YColorLed 216

S
save3DCalibration, YRefFrame 1272
saveSequence, YDisplay 445
saveToFlash, YModule 855
selectColorPen, YDisplayLayer 477
selectEraser, YDisplayLayer 478
selectFont, YDisplayLayer 479
selectGrayPen, YDisplayLayer 480
Sensor 1315
Sequence 336, 338, 350
SerialPort 1354
Servo 1411

set_adminPassword, YNetwork 946
set_allSettings, YModule 856
set_analogCalibration, YAnButton 148
set_autoStart, YDataLogger 329
set_autoStart, YWatchdog 1741
set_beacon, YModule 857
set_beaconDriven, YDataLogger 330
set_bearing, YRefFrame 1273
set_bitDirection, YDigitalIO 396
set_bitOpenDrain, YDigitalIO 397
set_bitPolarity, YDigitalIO 398
set_bitState, YDigitalIO 399
set_blinking, YLed 725
set_brakingForce, YMotor 896
set_brightness, YDisplay 446
set_calibrationMax, YAnButton 149
set_calibrationMin, YAnButton 150
set_callbackCredentials, YNetwork 947
set_callbackEncoding, YNetwork 948
set_callbackMaxDelay, YNetwork 949
set_callbackMethod, YNetwork 950
set_callbackMinDelay, YNetwork 951
set_callbackUrl, YNetwork 952
set_currentValue, YAltitude 107
set_cutOffVoltage, YMotor 897
set_discoverable, YNetwork 953
set_drivingForce, YMotor 898
set_dutyCycle, YPwmOutput 1144
set_dutyCycleAtPowerOn, YPwmOutput 1145
set_enabled, YDisplay 447
set_enabled, YHubPort 661
set_enabled, YPwmOutput 1146
set_enabled, YServo 1437
set_enabledAtPowerOn, YPwmOutput 1147
set_enabledAtPowerOn, YServo 1438
set_failSafeTimeout, YMotor 899
set_frequency, YMotor 900
set_frequency, YPwmOutput 1148
set_highestValue, YAccelerometer 67
set_highestValue, YAltitude 108
set_highestValue, YCarbonDioxide 186
set_highestValue, YCompass 255
set_highestValue, YCurrent 294
set_highestValue, YGenericSensor 576
set_highestValue, YGyro 632
set_highestValue, YHumidity 696
set_highestValue, YLightSensor 764
set_highestValue, YMagnetometer 807
set_highestValue, YPower 1021
set_highestValue, YPressure 1060
set_highestValue, YPwmInput 1107
set_highestValue, YQt 1208
set_highestValue, YSensor 1346
set_highestValue, YTemperature 1478
set_highestValue, YTilt 1518
set_highestValue, YVoc 1557
set_highestValue, YVoltage 1596
set_hours, YWakeUpSchedule 1698
set_hslColor, YColorLed 217
set_logFrequency, YAccelerometer 68
set_logFrequency, YAltitude 109
set_logFrequency, YCarbonDioxide 187
set_logFrequency, YCompass 256
set_logFrequency, YCurrent 295
set_logFrequency, YGenericSensor 577
set_logFrequency, YGyro 633
set_logFrequency, YHumidity 697
set_logFrequency, YLightSensor 765
set_logFrequency, YMagnetometer 808
set_logFrequency, YPower 1022
set_logFrequency, YPressure 1061
set_logFrequency, YPwmInput 1108
set_logFrequency, YQt 1209
set_logFrequency, YSensor 1347
set_logFrequency, YTemperature 1479
set_logFrequency, YTilt 1519
set_logFrequency, YVoc 1558
set_logFrequency, YVoltage 1597
set_logicalName, YAccelerometer 69
set_logicalName, YAltitude 110
set_logicalName, YAnButton 151
set_logicalName, YCarbonDioxide 188
set_logicalName, YColorLed 218
set_logicalName, YCompass 257
set_logicalName, YCurrent 296
set_logicalName, YDataLogger 331
set_logicalName, YDigitalIO 400
set_logicalName, YDisplay 448
set_logicalName, YDualPower 509
set_logicalName, YFiles 536
set_logicalName, YGenericSensor 578
set_logicalName, YGyro 634
set_logicalName, YHubPort 662
set_logicalName, YHumidity 698
set_logicalName, YLed 726
set_logicalName, YLightSensor 766
set_logicalName, YMagnetometer 809
set_logicalName, YModule 858
set_logicalName, YMotor 901
set_logicalName, YNetwork 954
set_logicalName, YOsControl 982
set_logicalName, YPower 1023
set_logicalName, YPressure 1062
set_logicalName, YPwmInput 1109
set_logicalName, YPwmOutput 1149
set_logicalName, YPwmPowerSource 1173
set_logicalName, YRealTimeClock 1238
set_logicalName, YRefFrame 1274
set_logicalName, YRelay 1307
set_logicalName, YSensor 1348
set_logicalName, YSerialPort 1400
set_logicalName, YServo 1439
set_logicalName, YTemperature 1480
set_logicalName, YTilt 1520
set_logicalName, YYVoc 1559
set_logicalName, YVoltage 1598
set_logicalName, YVSource 1631
set_logicalName, YYtExt 1661
set_logicalName, YYWakeupMonitor 1699
set_logicalName, YYWatchdog 1742
set_logicalName, YYWireless 1779
set_lowestValue, YAccelerometer 70
set_lowestValue, YAltitude 111
set_lowestValue, YCarbonDioxide 189
set_lowestValue, YCompass 258
set_lowestValue, YCurrent 297
set_lowestValue, YGenericSensor 579
set_lowestValue, YGyro 635
set_lowestValue, YHumidity 699
set_lowestValue, YLightSensor 767
set_lowestValue, YMagnetometer 810
set_lowestValue, YPressure 1024
set_lowestValue, YPower 1063
set_lowestValue, YPwmInput 1110
set_lowestValue, YQt 1211
set_lowestValue, YSensor 1349
set_lowestValue, YTemperature 1481
set_lowestValue, YTilt 1521
set_lowestValue, YYVoc 1560
set_lowestValue, YVoltage 1599
set_luminosity, YLed 727
set_luminosity, YModule 859
set_maxTimeOnStateA, YRelay 1308
set_maxTimeOnStateA, YWatchdog 1743
set_maxTimeOnStateB, YRelay 1309
set_maxTimeOnStateB, YWatchdog 1744
set_measureType, YLightSensor 768
set_minutes, YWakeUpSchedule 1700
set_minutesA, YWakeUpSchedule 1701
set_minutesB, YWakeUpSchedule 1702
set_monthDays, YWakeUpSchedule 1703
set_months, YWakeUpSchedule 1704
set_mountPosition, YRefFrame 1275
set_neutral, YServo 1440
set_nextWakeUp, YWakeUpMonitor 1662
set_orientation, YDisplay 449
set_output, YRelay 1310
set_output, YWatchdog 1745
set_outputVoltage, YDigitalIO 401
set_overCurrentLimit, YMotor 902
set_period, YPwmOutput 1150
set_portDirection, YDigitalIO 402
set_portOpenDrain, YDigitalIO 403
set_portPolarity, YDigitalIO 404
set_portState, YDigitalIO 405
set_position, YServo 1441
set_positionAtPowerOn, YServo 1442
set_power, YLed 728
set_powerControl, YDualPower 510
set_powerDuration, YWakeUpMonitor 1663
set_powerMode, YPwmPowerSource 1174
set_primaryDNS, YNetwork 955
set_protocol, YSerialPort 1401
set_pulseDuration, YPwmOutput 1151
set_pwmReportMode, YPwmPowerSource 1173
set_qnh, YAltitude 112
set_range, YServo 1443
set_recording, YDataLogger 332
set_reportFrequency, YAccelerometer 71
set_reportFrequency, YAltitude 113
set_reportFrequency, YCarbonDioxide 190
set_reportFrequency, YCompass 259
set_reportFrequency, YCurrent 298
set_reportFrequency, YGenericSensor 580
set_reportFrequency, YGyro 636
set_reportFrequency, YHumidity 700
set_reportFrequency, YLightSensor 769
set_reportFrequency, YMagnetometer 811
set_reportFrequency, YPower 1025
set_reportFrequency, YPressure 1064
set_reportFrequency, YPwmInput 1112
set_reportFrequency, YYVoc 1561
set_reportFrequency, YVoltage 1600
set_resolution, YAccelerometer 72
set_resolution, YAltitude 114
set_resolution, YCarbonDioxide 191
set_resolution, YCompass 260
set_resolution, YCurrent 299
set_resolution, YGenericSensor 581
set_resolution, YGyro 637
set_resolution, YHumidity 701
set_resolution, YLightSensor 770
set_resolution, YMagnetometer 812
set_resolution, YPower 1026
set_resolution, YPressure 1065
set_resolution, YPwmInput 1113
set_resolution, YYVoc 1562
set_resolution, YVoltage 1601
set_rgbColor, YColorLed 219
set_rgbColorAtPowerOn, YColorLed 220
set_RTS, YSerialPort 1399
set_running, YWatchdog 1746
set_secondaryDNS, YNetwork 956
set_sensitivity, YAnButton 152
set_sensorType, YTemperature 1484
set_serialMode, YSerialPort 1402
set_signalBias, YGenericSensor 582
set_signalRange, YGenericSensor 583
set_sleepCountdown, YWakeUpMonitor 1664
set_starterTime, YMotor 903
set_startupSeq, YDisplay 450
set_state, YRelay 1311
set_state, YWatchdog 1747
set_stateAtPowerOn, YRelay 1312
set_stateAtPowerOn, YWatchdog 1748
set_timeUTC, YDataLogger 333
set_triggerDelay, YWatchdog 1749
set_triggerDuration, YWatchdog 1750
set_unit, YGenericSensor 584
set_unixTime, YRealTimeClock 1239
set_userData, YAccelerometer 73
set_userData, YAltitude 115
set_userData, YAnButton 153
set_userData, YCarbonDioxide 192
set_userData, YColorLed 221
set_userData, YCompass 261
set_userData, YCurrent 300
set_userData, YDataLogger 334
set_userData, YDigitalIO 406
set_userData, YDisplay 451
set_userData, YDualPower 511
set_userData, YFiles 537
set_userData, YGenericSensor 585
set_userData, YGyro 638
set_userData, YHubPort 663
set_userData, YHumidity 702
set_userData, YLed 729
set_userData, YLightSensor 771
set_userData, YMagnetometer 813
set_userData, YModule 860
set_userData, YMotor 904
set_userData, YNetwork 957
set_userData, YOsControl 983
set_userData, YPower 1027
set_userData, YPressure 1066
set_userData, YPWMInput 1114
set_userData, YPWMOutput 1152
set_userData, YPWMSource 1175
set_userData, YQ1 1214
set_userData, YRealTimeClock 1240
set_userData, YRefFrame 1276
set_userData, YRelay 1313
set_userData, YSensor 1352
set_userData, YSerialPort 1403
set_userData, YServo 1444
set_userData, YTemperature 1485
set_userData, YTilt 1524
set_userData, YVoc 1563
set_userData, YVoltage 1602
set_userData, YVSource 1632
set_userData, YWakeUpMonitor 1665
set_userData, YWakeUpSchedule 1705
set_userData, YWatchdog 1751
set_userData, YWireless 1780
set_userPassword, YNetwork 958
set_userVar, YModule 861
set_utcOffset, YRealTimeClock 1241
set_valueRange, YGenericSensor 586
set_voltage, YVSource 1633
set_weekDays, YWakeUpSchedule 1706
set_wwwWatchdogDelay, YNetwork 959
setAntialiasingMode, YDisplayLayer 481
setConsoleBackground, YDisplayLayer 482
setConsoleMargins, YDisplayLayer 483
setConsoleWordWrap, YDisplayLayer 484
setLayerPosition, YDisplayLayer 485
shutdown, YOsControl 984
Sleep, YAPI 28
sleep, YWakeUpMonitor 1666
sleepFor, YWakeUpMonitor 1667
sleepUntil, YWakeUpMonitor 1668
softAPNetwork, YWireless 1781
Source 1604
start3DCalibration, YRefFrame 1277
stopSequence, YDisplay 452
Supply 488
swapLayerContent, YDisplay 453

T
Temperature 1446
Tilt 1487
Time 1216
toggle_bitState, YDigitalIO 407
triggerFirmwareUpdate, YModule 862
TriggerHubDiscovery, YAPI 29

U
Unformatted 350
unhide, YDisplayLayer 486
UnregisterHub, YAPI 30
UpdateDeviceList, YAPI 31
updateFirmware, YModule 863
upload, YDisplay 454
upload, YFiles 538
useDHCP, YNetwork 960
useStaticIP, YNetwork 961

V
Value 815
Visual 3
Voltage 1565, 1604
toltageMove, YVSource 1634

W
wakeUp, YWakeUpMonitor 1669
WakeUpMonitor 1636
WriteArray, YSerialPort 1404
writeBin, YSerialPort 1405
writeHex, YSerialPort 1406
writeLine, YSerialPort 1407
writeMODBUS, YSerialPort 1408
writeStr, YSerialPort 1409

Y

YAccelerometer 35-73
YAltitude 77-115
YAnButton 119-153
YAPI 14-31
YCarbonDioxide 157-192
yCheckLogicalName 14
YColorLed 195-221
YCompass 225-261
YCurrent 265-300
YDataLogger 304-334
YDataRun 336
YDataSet 339-348
YDataStream 351-363
YDigitalIO 367-407
yDisableExceptions 15
YDisplay 411-454
YDisplayLayer 457-486
YDualPower 489-511
yEnableExceptions 16
YFiles 514-538
yFindAccelerometer 35
yFindAltitude 77
yFindAnButton 119
yFindCarbonDioxide 157
yFindColorLed 195
yFindCompass 225
yFindCurrent 265
yFindDataLogger 304
yFindDigitalIO 367
yFindDisplay 411
yFindDualPower 489
yFindFiles 514
yFindGenericSensor 542
yFindGyro 591
yFindHubPort 641
yFindHumidity 667
yFindLed 705
yFindLightSensor 733
yFindMagnetometer 775
yFindModule 823
yFindMotor 867
yFindNetwork 909
yFindOsControl 964
yFindPower 988
yFindPressure 1031
yFindPwmInput 1070
yFindPwmOutput 1118
yFindPwmPowerSource 1155

yFindQt 1179
yFindRealTimeClock 1217
yFindRefFrame 1245
yFindRelay 1281
yFindSensor 1317
yFindSerialPort 1357
yFindServo 1413
yFindTemperature 1448
yFindTilt 1489
yFindVoc 1528
yFindVoltage 1567
yFindVSource 1605
yFindWakeUpMonitor 1638
yFindWakeUpSchedule 1673
yFindWatchdog 1710
yFindWireless 1754
yFirstAccelerometer 36
yFirstAltitude 78
yFirstAnButton 120
yFirstCarbonDioxide 158
yFirstColorLed 196
yFirstCompass 226
yFirstCurrent 266
yFirstDataLogger 305
yFirstDigitalIO 368
yFirstDisplay 412
yFirstDualPower 490
yFirstFiles 515
yFirstGenericSensor 543
yFirstGyro 592
yFirstHubPort 642
yFirstHumidity 668
yFirstLed 706
yFirstLightSensor 734
yFirstMagnetometer 776
yFirstModule 824
yFirstMotor 868
yFirstNetwork 910
yFirstOsControl 965
yFirstPower 989
yFirstPressure 1032
yFirstPwmInput 1071
yFirstPwmOutput 1119
yFirstPwmPowerSource 1156
yFirstQt 1180
yFirstRealTimeClock 1218
yFirstRefFrame 1246
yFirstRelay 1282
yFirstSensor 1318
yFirstSerialPort 1358
yFirstServo 1414
yFirstTemperature 1449
yFirstTilt 1490
yFirstVoc 1529
yFirstVoltage 1568
yFirstVSource 1606
yFirstWakeUpMonitor 1639
yFirstWakeUpSchedule 1674
yFirstWatchdog 1711