Java for Android API Reference
Table of contents

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

2. Using the Yocto-Demo with Android ..................................................................................... 3
   2.1. Native access and VirtualHub ....................................................................................... 3
   2.2. Getting ready .................................................................................................................. 3
   2.3. Compatibility .................................................................................................................. 3
   2.4. Activating the USB port under Android ........................................................................ 4
   2.5. Control of the Led function ........................................................................................... 6
   2.6. Control of the module part ............................................................................................. 8
   2.7. Error handling ............................................................................................................... 13
   Blueprint ................................................................................................................................. 16

3. Reference .................................................................................................................................. 16
   3.1. General functions ......................................................................................................... 17
   3.2. Accelerometer function interface .................................................................................. 36
   3.3. Altitude function interface ............................................................................................ 78
   3.4. AnButton function interface ....................................................................................... 120
   3.5. CarbonDioxide function interface ................................................................................. 158
   3.6. ColorLed function interface ......................................................................................... 197
   3.7. Compass function interface .......................................................................................... 226
   3.8. Current function interface ............................................................................................. 266
   3.9. DataLogger function interface ....................................................................................... 305
   3.10. Formatted data sequence ............................................................................................. 339
   3.11. Recorded data sequence ............................................................................................... 349
   3.12. Unformatted data sequence ......................................................................................... 361
   3.13. Digital IO function interface ....................................................................................... 376
   3.14. Display function interface ............................................................................................ 420
   3.15. DisplayLayer object interface ..................................................................................... 467
   3.16. External power supply control interface ...................................................................... 499
   3.17. Files function interface ............................................................................................... 524
   3.18. GenericSensor function interface ................................................................................. 551
   3.19. Gyroscope function interface ..................................................................................... 600
   3.20. Yocto-hub port interface ............................................................................................. 651
   3.21. Humidity function interface ......................................................................................... 676
   3.22. Led function interface ................................................................................................. 715
| 3.23. LightSensor function interface | ............................................................... | 742 |
| 3.24. Magnetometer function interface | ............................................................... | 784 |
| 3.25. Measured value | ............................................................... | 826 |
| 3.26. Module control interface | ............................................................... | 832 |
| 3.27. Motor function interface | ............................................................... | 872 |
| 3.28. Network function interface | ............................................................... | 913 |
| 3.29. OS control | ............................................................... | 970 |
| 3.30. Power function interface | ............................................................... | 993 |
| 3.31. Pressure function interface | ............................................................... | 1036 |
| 3.32. PwmInput function interface | ............................................................... | 1075 |
| 3.33. Pwm function interface | ............................................................... | 1123 |
| 3.34. PwmPowerSource function interface | ............................................................... | 1161 |
| 3.35. Quaternion interface | ............................................................... | 1184 |
| 3.36. Real Time Clock function interface | ............................................................... | 1223 |
| 3.37. Reference frame configuration | ............................................................... | 1250 |
| 3.38. Relay function interface | ............................................................... | 1286 |
| 3.39. Sensor function interface | ............................................................... | 1322 |
| 3.40. SerialPort function interface | ............................................................... | 1361 |
| 3.41. Servo function interface | ............................................................... | 1418 |
| 3.42. Temperature function interface | ............................................................... | 1453 |
| 3.43. Tilt function interface | ............................................................... | 1494 |
| 3.44. Voc function interface | ............................................................... | 1533 |
| 3.45. Voltage function interface | ............................................................... | 1572 |
| 3.46. Voltage source function interface | ............................................................... | 1611 |
| 3.47. WakeUpMonitor function interface | ............................................................... | 1643 |
| 3.48. WakeUpSchedule function interface | ............................................................... | 1678 |
| 3.49. Watchdog function interface | ............................................................... | 1715 |
| 3.50. Wireless function interface | ............................................................... | 1760 |

Index | ........................................................................................................ | 1791 |
1. Introduction

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

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

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

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

2.1. Native access and VirtualHub

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

2.2. Getting ready

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

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

2.3. Compatibility

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

---

\(^1\) www.yoctopuce.com/EN/libraries.php
Android 4.x

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

USB host support

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

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

Supported hardware

The library is tested and validated on the following machines:

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

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

2.4. Activating the USB port under Android

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

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

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

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

---

2 Yoctohubs are a plug and play way to add network connectivity to your Yoctopuce devices. more info on http://www.yoctopuce.com/EN/products/category/extensions-and-networking
it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, your must provide a pointer on the application context by calling the EnableUSBHost method of the YAPI class before the first USB access. This function takes as arguments an object of the android.content.Context class (or of a subclass). As the Activity class is a subclass of Context, it is simpler to call YAPI.EnableUSBHost(this); in the method onCreate of your application. If the object passed as parameter is not of the correct type, a YAPI_Exception exception is generated.

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

**Autorun**

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

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

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

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <usb-device vendor-id="9440" product-id="12"/>
    <usb-device vendor-id="9440" product-id="13"/>
</resources>
```
2.5. Control of the Led function

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

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

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

Let us look at these lines in more details.

**YAPI.EnableUSBHost**

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

**YAPI.RegisterHub**

The YAPI.RegisterHub function initializes the Yoctopuce API and indicates where the modules should be looked for. The parameter is the address of the virtual hub able to see the devices. If the string "usb" is passed as parameter, the API works with modules locally connected to the machine. If the initialization does not succeed, an exception is thrown.

**YLed.FindLed**

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

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

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

**isOnline**

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

**set_power**

The set_power() function of the objet returned by YLed.FindLed allows you to turn on and off the led. The argument is YLed.POWER_ON or YLed.POWER_OFF. In the reference on the programming interface, you will find more methods to precisely control the luminosity and make the led blink automatically.
A real example
Launch your Java environment and open the corresponding sample project provided in the directory Examples//Doc-Examples of the Yoctopuce library.

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

```java
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YLed;

public class GettingStarted_Yocto_Demo extends Activity implements OnItemSelectedListener {
    private YLed led = null;
    private ArrayAdapter<String> aa;

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

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

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

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id) {
        String hwid = parent.getItemAtPosition(pos).toString();
        led = YLed.FindLed(hwid);
    }
}
```
2.6. Control of the module part

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

```java
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;
import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

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

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.modulecontrol);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart() {
        // Do something in response to button click
        if (led != null) {
            try {
                led.setPower(YLed.POWER_ON);
            } catch (YAPI_Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```
super.onStart();

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

// refresh Spinner with detected relay
aa.notifyDataSetChanged();

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

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

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

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

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

public void toggleBeacon(View view) {
    if (module == null)
        return;
    boolean on = ((Switch) view).isChecked();
Each property `xxx` of the module can be read thanks to a method of type `YModule.get_xxxx()`, and properties which are not read-only can be modified with the help of the `YModule.set_xxx()` method. For more details regarding the used functions, refer to the API chapters.

Changing the module settings

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

```java
package com.yoctopuce.doc_examples;

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

public class SaveSettings extends Activity implements OnItemSelectedListener {

    private ArrayAdapter<String> aa;
    private YModule module = null;

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

    @Override
    protected void onStart() {
        super.onStart();
        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule r = YModule.FirstModule();
            while (r != null) {
                String hwid = r.get_hardwareId();
                aa.add(hwid);
            }
        }
    }
}
```
```java
r = r.nextModule();
}
} catch (YAPI_Exception e) {

e.printStackTrace();
}
// refresh Spinner with detected relay
aa.notifyDataSetChanged();

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

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

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

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

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

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

```java
package com.yoctopuce.doc_examples;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.LinearLayout;
import android.widget.TextView;
import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class Inventory extends Activity {

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

    public void refreshInventory(View view) {
        LinearLayout layout = (LinearLayout) findViewById(R.id.inventoryList);
        layout.removeAllViews();
        try {
            YAPI.UpdateDeviceList();
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() + 
                "\n";
                TextView tx = new TextView(this);
                tx.setText(line);
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

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

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

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

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

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

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

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

```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 *
```

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

<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 network 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 <code>RegisterHub</code>.</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()

YAPI.CheckLogicalName() checks if a given string is valid as logical name for a module or a function.

boolean CheckLogicalName( String name)

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

Parameters:
  name a string containing the name to check.

Returns:
  true if the name is valid, false otherwise.
YAPI.EnableUSBHost()

This function is used only on Android.

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

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

**Parameters:**
- `osContext` an object of class `android.content.Context` (or any subclass).
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()

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

```c
long GetTickCount()
```

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

**Returns**:

a long integer corresponding to the millisecond counter.
YAPI.HandleEvents()  
yHandleEvents() YAPI.HandleEvents()

Maintains the device-to-library communication channel.

```
int HandleEvents( )
```

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

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

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

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

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

Initializes the Yoctopuce programming library explicitly.

```c
int InitAPI( int mode)
```

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

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

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

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

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

YAPI.PreregisterHub() is a fault-tolerant alternative to RegisterHub().

```
int PreregisterHub( String url)
```

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

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

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

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

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

```c
void RegisterDeviceArrivalCallback(DeviceArrivalCallback arrivalCallback)
```

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

**Parameters:**
- `arrivalCallback` a procedure taking a `YModule` parameter, or `null`
Register a callback function, to be called each time a device is unplugged.

```c
void RegisterDeviceRemovalCallback( DeviceRemovalCallback removalCallback)
```

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

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

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

```c
int RegisterHub( String url)
```

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

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

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

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

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

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

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

```c
void RegisterHubDiscoveryCallback( HubDiscoveryCallback hubDiscoveryCallback)
```

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

**Parameters :**
- `hubDiscoveryCallback` a procedure taking two string parameter, or null
Registers a log callback function.

```c
void RegisterLogFunction(LogCallback logfun)
```

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

**Parameters:**
- `logfun` a procedure taking a string parameter, or `null`
YAPI.Sleep()  
ySleep()  
YAPI.Sleep()

Pauses the execution flow for a specified duration.

```c
int Sleep( long ms_duration)
```

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

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

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

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

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

**yTriggerHubDiscovery()**

YAPI.TriggerHubDiscovery()

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

```c
int TriggerHubDiscovery( )
```

**Parameters:**

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

**Returns:**

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

YAPI.UnregisterHub() yUnregisterHub()

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

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

**Parameters:**
- `url` a string containing either "usb" or the
YAPI.UpdateDeviceList()
yUpdateDeviceList() YAPI.UpdateDeviceList ()

Triggers a (re)detection of connected Yoctopuce modules.

```c
int UpdateDeviceList ( )
```

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

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

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

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

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

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;
```

```php
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.
### 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 acceleration 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 accelerometer.</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_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>
</tbody>
</table>
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>accelerometer.set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>accelerometer.set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>accelerometer.set_logicalName(newval)</code></td>
<td>Changes the logical name of the accelerometer.</td>
</tr>
<tr>
<td><code>accelerometer.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>accelerometer.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>accelerometer.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>accelerometer.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.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>
YAccelerometer.FindAccelerometer()

`YAccelerometer.FindAccelerometer()` retrieves an accelerometer for a given identifier.

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.

YAccelerometer FirstAccelerometer()

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

Returns:
- A pointer to a YAccelerometer object, corresponding to the first accelerometer currently online, or a null pointer if there are none.
accelerometer→calibrateFromPoints()  
accelerometer.calibrateFromPoints()

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

```java
int calibrateFromPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

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

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

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

accelerometer.describe()

accelerometer.describe()

Returns a short text that describes unambiguously the instance of the accelerometer in the form 

TYPE(NAME)=SERIAL.FUNCTIONID.

String describe()

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

Returns:

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

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

String get_advertisedValue()  

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
accelerometer→get_currentRawValue()  
accelerometer→currentRawValue()  
accelerometer.get_currentRawValue()

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

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

YAccelerometer

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

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

double get_currentValue() 

Returns:

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
accelerometer→get_errorMessage()  
accelerometer→errorMessage()  
accelerometer.get_errorMessage()  

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

String get_errorMessage()  

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

**Returns:**

a string corresponding to the latest error message that occurred while using the accelerometer object
accelerometer.get_errorType()
accelerometer.getErrorType()
accelerometer.get_errorType()

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

int get_errorType()

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

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

**accelerometer.get_friendlyName()**

**accelerometer.friendlyName()**

**accelerometer.get_friendlyName()**

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

```java
String get_friendlyName()
```

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

**Returns:**

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

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
accelerometer.get_functionDescriptor()

YAccelerometer
accelerometer.functionDescriptor()
accelerometer.get_functionDescriptor()

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

String get_functionDescriptor()

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

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
Returns the hardware identifier of the accelerometer, without reference to the module.

**String get_functionId( )**

For example *relay1*

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

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

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

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

String get_hardwareId()

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

Returns :

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
YAccelerometer
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.
3. Reference

YAccelerometer

accelerometer → get_logicalName()
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.
3. Reference

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

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

YModule `get_module()`

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

**Returns:**

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

\[
\text{YDataSet } \text{get}_{\text{recordedData}}( \text{long } \text{startTime}, \text{long } \text{endTime}) \]

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.
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 Y_REPORTFREQUENCY_INVALID.
accelerometer\.get\_resolution() \hspace{1em} \text{YAccelerometer}

accelerometer\rightarrow resolution() \hspace{1em} \text{accelerometer.get\_resolution()}

Returns the resolution of the measured values.

\begin{verbatim}
double get\_resolution() 
\end{verbatim}

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 Y\_RESOLUTION\_INVALID.
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`. 
YAccelerometer

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

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

Object `get_userData()`

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

Returns:
the object stored previously by the caller.
accelerometer→get_xValue()
accelerometer→xValue()
accelerometer.get_xValue()  

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

<table>
<thead>
<tr>
<th>double get_xValue( )</th>
</tr>
</thead>
</table>

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

```java
double get_yValue()
```

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

On failure, throws an exception or returns Y_YVALUE_INVALID.
accelerometer→get_zValue()  \hspace{1cm} \text{YAccelerometer}
accelerometer→zValue()
accelerometer.get_zValue()

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

double get_zValue( )

\textbf{Returns :}
\begin{itemize}
  \item a floating point number corresponding to the Z component of the acceleration, as a floating point number
\end{itemize}

On failure, throws an exception or returns \texttt{Y\_ZVALUE\_INVALID}.
accelerometer → isOnline()

accelerometer.isOnline()

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

boolean isOnline()

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

**Returns:**

true if the accelerometer can be reached, and false otherwise
Preloads the accelerometer 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.
accelerometer.loadCalibrationPoints() retrieves error correction data points previously entered using the method calibrateFromPoints.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

On failure, throws an exception or returns a negative error code.
accelerometer\rightarrow\textit{nextAccelerometer()}

\texttt{accelerometer.nextAccelerometer()}

Continues the enumeration of accelerometers started using \texttt{yFirstAccelerometer()}.  

\texttt{YAccelerometer \textit{nextAccelerometer}()}

\textbf{Returns :}

\begin{itemize}
  \item a pointer to a \texttt{YAccelerometer} object, corresponding to an accelerometer currently online, or a \texttt{null} pointer if there are no more accelerometers to enumerate.
\end{itemize}
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.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

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

`accelerometer->set_highestValue()`

`accelerometer->setHighestValue()`

`accelerometer.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.
**accelerometer → set_logFrequency()**  
**accelerometer → setLogFrequency()**  
**accelerometer.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```java
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 (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.
accelerometer.set_logicalName()

Changes the logical name of the accelerometer.

int set_logicalName(String newval)

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

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

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

On failure, throws an exception or returns a negative error code.
accelerometer->set_lowestValue()
accelerometer->setLowestValue()
accelerometer.set_lowestValue()

Changes the recorded minimal value observed.

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

**Parameters:**
- `newval` a floating point number corresponding to the recorded minimal value observed

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
accelerometer → set_reportFrequency()
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.
accelerometer→set_resolution()
accelerometer→setResolution()
accelerometer.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.
accelerometer→set_userData()
accelerometer→setUserData()
accelerometer.set_userData()

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

```javascript
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

3.3. 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
#include "yocto_altitude.h"
```
```pas
uses yocto_altitude;
```
```vb
yocto_altitude.vb
```
```cs
yocto_altitude.cs
```
```java
import com.yoctopuce.YoctoAPI.YAltitude;
```
```py
from yocto_altitude import *
```

### Global functions

- **yFindAltitude(func)**
  
  Retrieves an altimeter for a given identifier.

- **yFirstAltitude()**
  
  Starts the enumeration of altimeters currently accessible.

### YAltitude methods

- **altitude→calibrateFromPoints(rawValues, refValues)**
  
  Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

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

- **altitude→get_advertisedValue()**
  
  Returns the current value of the altimeter (no more than 6 characters).

- **altitude→get_currentRawValue()**
  
  Returns the uncalibrated, unrounded raw value returned by the sensor, in meters, as a floating point number.

- **altitude→get_currentValue()**
  
  Returns the current value of the altitude, in meters, as a floating point number.

- **altitude→get_errorMessage()**
  
  Returns the error message of the latest error with the altimeter.

- **altitude→get_errorType()**
  
  Returns the numerical error code of the latest error with the altimeter.

- **altitude→get_friendlyName()**
  
  Returns a global identifier of the altimeter in the format MODULE_NAME.FUNCTION_NAME.

- **altitude→get_functionDescriptor()**
  
  Returns a unique identifier of type YFUN_DESCR corresponding to the function.

- **altitude→get_functionId()**
  
  Returns the hardware identifier of the altimeter, without reference to the module.

- **altitude→get_hardwareId()**
  
  Returns the unique hardware identifier of the altimeter in the form SERIAL.FUNCTIONID.
### 3. Reference

#### altitude→get_highestValue()
- Returns the maximal value observed for the altitude since the device was started.

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

#### altitude→get_logicalName()
- Returns the logical name of the altimeter.

#### altitude→get_lowestValue()
- Returns the minimal value observed for the altitude since the device was started.

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

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

#### altitude→get_qnh()
- Returns the barometric pressure adjusted to sea level used to compute the altitude (QNH).

#### altitude→get_recordedData(startTime, endTime)
- Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

#### altitude→get_reportFrequency()
- Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

#### altitude→get_resolution()
- Returns the resolution of the measured values.

#### altitude→get_unit()
- Returns the measuring unit for the altitude.

#### altitude→get_userData()
- Returns the value of the userData attribute, as previously stored using method set_userData.

#### altitude→isOnline()
- Checks if the altimeter is currently reachable, without raising any error.

#### altitude→isOnline_async(callback, context)
- Checks if the altimeter is currently reachable, without raising any error (asynchronous version).

#### altitude→load(msValidity)
- Preloads the altimeter cache with a specified validity duration.

#### altitude→loadCalibrationPoints(rawValues, refValues)
- Retrieves error correction data points previously entered using the method calibrateFromPoints.

#### altitude→load_async(msValidity, callback, context)
- Preloads the altimeter cache with a specified validity duration (asynchronous version).

#### altitude→nextAltitude()
- Continues the enumeration of altimeters started using yFirstAltitude().

#### altitude→registerTimedReportCallback(callback)
- Registers the callback function that is invoked on every periodic timed notification.

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

#### altitude→set_currentValue(newval)
- Changes the current estimated altitude.

#### altitude→set_highestValue(newval)
- Changes the recorded maximal value observed.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>altitude-&gt;set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>altitude-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the altimeter.</td>
</tr>
<tr>
<td><code>altitude-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>altitude-&gt;set_qnh(newval)</code></td>
<td>Changes the barometric pressure adjusted to sea level used to compute the altitude (QNH).</td>
</tr>
<tr>
<td><code>altitude-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>altitude-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>altitude-&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>altitude-&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>
YAltitude.FindAltitude()

YAltitude.FindAltitude()

Retrieves an altimeter for a given identifier.

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

Starts the enumeration of altimeters currently accessible.

YAltitude FirstAltitude()

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

Returns:
- a pointer to a YAltitude object, corresponding to the first altimeter currently online, or a null pointer if there are none.
### calibrateFromPoints()

`altitude.calibrateFromPoints()`

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

```java
int calibrateFromPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

#### Parameters:

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

#### Returns:

- `YAPI_SUCCESS` if the call succeeds.

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

altitude.describe()

Returns a short text that describes unambiguously the instance of the altimeter 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 altimeter (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
### Reference

**YAltitude**

#### altitude→get_advertisedValue()

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

String `get_advertisedValue()`

**Returns:**

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

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


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

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

YAltitude

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

```java
double get_currentValue()
```

Returns:

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
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
altitude\rightarrow get\_errorType()
altitude\rightarrow errorType()\texttt{altitude.get\_errorType()}

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

\textbf{int} \texttt{get\_errorType()}\n
This method is mostly useful when using the Yoctopuce library with exceptions disabled.

\textbf{Returns}:

\textit{a number corresponding to the code of the latest error that occurred while using the altimeter object}
YAltitude

altitude.get_friendlyName()

altitude.friendlyName()

altitude.get_friendlyName()

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

\begin{verbatim}
String get_friendlyName()
\end{verbatim}

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

\begin{verbatim}
Returns :
\begin{itemize}
\item a string that uniquely identifies the altimeter using logical names (ex: MyCustomName.relay1)
\end{itemize}
\end{verbatim}

On failure, throws an exception or returns \texttt{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.

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>an identifier of type <code>YFUN_DESCR</code>.</td>
</tr>
</tbody>
</table>

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

**altitude.get_functionId()**

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

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

Returns the unique hardware identifier of the altimeter in the form SERIALFUNCTIONID.

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.
### 3. Reference

YAltitude

#### altitude→get_highestValue()  
#### altitude→highestValue()  
#### altitude.get_highestValue()  

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

<table>
<thead>
<tr>
<th>double get_highestValue( )</th>
</tr>
</thead>
</table>

**Returns:**  
A floating point number corresponding to the maximal value observed for the altitude since the device was started.

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
altitude→get_logFrequency()
altitude→logFrequency()
altitude.get_logFrequency()

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

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.
YAltitude

```java
altitude.get_logicalName()
```

Returns the logical name of the altimeter.

```java
String get_logicalName()
```

**Returns:**

- a string corresponding to the logical name of the altimeter.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
altitude\rightarrow get\_lowestValue() 
altitude\rightarrow lowestValue() 
altitude.\texttt{get\_lowestValue}()

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

\textbf{double get\_lowestValue()}

Returns:

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

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

### altitude.get_module()
gets_module()

**YModule**

<table>
<thead>
<tr>
<th>YModule get_module()</th>
</tr>
</thead>
</table>

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

**Returns:**

- an instance of **YModule**
YAltitude

```
altitude->get_qnh()
```

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.

```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.

**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.

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 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.
3. Reference

YAltitude

altitude.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.
altitude→isOnline() → altitude.isOnline()

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

boolean 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
## Reference

### `altitude.load()`

Preloads the altimeter cache with a specified validity duration.

```c
int load( long msValidity)
```

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

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

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

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

3. Reference

altitude.loadCalibrationPoints()

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

```java
int loadCalibrationPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

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

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.
YAltitude

altitude.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

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

```java
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.
altitude\rightarrow\text{set\_lowestValue()}
altitude\rightarrow\text{setLowestValue()}
al\text{ltitude} . \text{set\_lowestValue()}

Changes the recorded minimal value observed.

\textbf{int set\_lowestValue( double newval)}

\textbf{Parameters :}
- \textbf{newval} a floating point number corresponding to the recorded minimal value observed

\textbf{Returns :}
- \textbf{YAPI\_SUCCESS} if the call succeeds.
- On failure, throws an exception or returns a negative error code.
Changes the barometric pressure adjusted to sea level used to compute the altitude (QNH).

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.
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.

Parameters:
- `newval` a floating point number corresponding to the resolution of the measured physical values

Returns:
- `YAPI_SUCCESS` if the call succeeds.

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

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. Reference

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 used 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
yocto_anbutton.cs
```
```java
import com.yoctopuce.YoctoAPI.YAnButton;
```
```py
from yocto_anbutton import *
```

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindAnButton(func)</code></td>
<td>Retrieves an analog input for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstAnButton()</code></td>
<td>Starts the enumeration of analog inputs currently accessible.</td>
</tr>
</tbody>
</table>

### YAnButton methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anbutton.describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the analog input in the form TYPE(NAME)=SERIAL.functionID.</td>
</tr>
<tr>
<td><code>anbutton.get_advertisedValue()</code></td>
<td>Returns the current value of the analog input (no more than 6 characters).</td>
</tr>
<tr>
<td><code>anbutton.get_analogCalibration()</code></td>
<td>Tells if a calibration process is currently ongoing.</td>
</tr>
<tr>
<td><code>anbutton.get_calibratedValue()</code></td>
<td>Returns the current calibrated input value (between 0 and 1000, included).</td>
</tr>
<tr>
<td><code>anbutton.get_calibrationMax()</code></td>
<td>Returns the maximal value measured during the calibration (between 0 and 4095, included).</td>
</tr>
<tr>
<td><code>anbutton.get_calibrationMin()</code></td>
<td>Returns the minimal value measured during the calibration (between 0 and 4095, included).</td>
</tr>
<tr>
<td><code>anbutton.get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the analog input.</td>
</tr>
<tr>
<td><code>anbutton.get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the analog input.</td>
</tr>
<tr>
<td><code>anbutton.get_friendlyName()</code></td>
<td>Returns a global identifier of the analog input in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><code>anbutton.get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
</tbody>
</table>
anbutton→get_functionId()
Returns the hardware identifier of the analog input, without reference to the module.

anbutton→get_hardwareId()
Returns the unique hardware identifier of the analog input in the form SERIAL.FUNCTIONID.

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

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

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

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

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

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

anbutton→get_pulseCounter()
Returns the pulse counter value

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

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

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

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

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

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

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

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

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

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

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

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

anbutton→set_calibrationMax(newval)
### 3. Reference

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>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>set_logicalName(newval)</code></td>
<td>Changes the logical name of the analog input.</td>
</tr>
<tr>
<td><code>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>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>
YAnButton.FindAnButton()

YAnButton.FindAnButton(String func)

Retrieves an analog input for a given identifier.

Parameters:

- `func`: a string that uniquely characterizes the analog input

Returns:

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

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.
**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.
YAnButton

anbutton.describe()  Returns a short text that describes unambiguously the instance of the analog input in the form TYPE(NAME)=SERIALFUNCTIONID.

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

YAnButton

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

Tells if a calibration process is currently ongoing.

int get_analogCalibration( )

Returns:

either Y_ANALOGCALIBRATION_OFF or Y_ANALOGCALIBRATION_ON

On failure, throws an exception or returns Y_ANALOGCALIBRATION_INVALID.
YAnButton

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

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

<table>
<thead>
<tr>
<th>int get_calibratedValue( )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Returns:</strong></td>
</tr>
<tr>
<td>an integer corresponding to the current calibrated input value (between 0 and 1000, included)</td>
</tr>
<tr>
<td>On failure, throws an exception or returns Y_CALIBRATEDVALUE_INVALID.</td>
</tr>
</tbody>
</table>
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.
3. Reference

YAnButton

anbutton→get_calibrationMin()
anbutton→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
YAnButton

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

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

```
int get_errorType()
```

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

**Returns:**

a number corresponding to the code of the latest error that occurred while using the analog input object
anbutton→get_friendlyName()
anbutton→friendlyName()
anbutton.get_friendlyName()

YAnButton

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

String get_friendlyName()

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

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

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

String get_functionDescriptor()

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

Returns:
(an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
anbutton→get_functionId()
anbutton→functionId()
anbutton.get_functionId()

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

String get_functionId()

For example relay1

**Returns:**

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

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
YAnButton

3. Reference

```java
anbutton→get_hardwareId()
anbutton→hardwareId()
anbutton.get_hardwareId()
```

**Returns**

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`. 


YAnButton

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

<table>
<thead>
<tr>
<th>Returns:</th>
<th>either \texttt{Y_ISPRESSED_FALSE} or \texttt{Y_ISPRESSED_TRUE}, according to true if the input (considered as binary) is active (closed contact), and false otherwise</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>On failure, throws an exception or returns \texttt{Y_ISPRESSED_INVALID}.</td>
</tr>
</tbody>
</table>
anbutton → get_lastTimePressed()
anbutton → lastTimePressed()
anbutton.get_lastTimePressed()

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

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

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

long get_lastTimeReleased( )

**Returns:**

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

On failure, throws an exception or returns Y_LASTTIMERELEASED_INVALID.
Returns the logical name of the analog input.

String get_logicalName()

**Returns:**

A string corresponding to the logical name of the analog input.

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

```python
anbutton→get_module()
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 online.

**Returns:**

an instance of YModule
Returns the pulse counter value

\[ \text{long } \text{get\_pulseCounter()} \]

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

On failure, throws an exception or returns \( \text{Y\_PULSECOUNTER\_INVALID} \).
anbutton\rightarrow get\_pulseTimer()
anbutton\rightarrow pulseTimer()
anbutton.\texttt{get\_pulseTimer}()

Returns the timer of the pulses counter (ms)

\begin{tabular}{|c|}
\hline
\texttt{long get\_pulseTimer( )} \\
\hline
\end{tabular}

\begin{itemize}
\item \textbf{Returns}:
\begin{quote}
an integer corresponding to the timer of the pulses counter (ms)
\end{quote}
\item On failure, throws an exception or returns \texttt{Y\_PULSETIMER\_INVALID}.
\end{itemize}
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.
YAnButton

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

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

```c
int get_sensitivity( )
```

Returns:

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

On failure, throws an exception or returns Y_SENSITIVITY_INVALID.
<table>
<thead>
<tr>
<th>YAnButton anbutton → get_userData()</th>
<th>anbutton → userData() anbutton.get_userData()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
<td></td>
</tr>
</tbody>
</table>

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.
The `YAnButton` class provides a method `isOnline()` to check if the analog input is currently reachable, without raising any error.

```java
boolean isOnline()
```

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

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

anbutton->load() anbutton::load()

Preloads the analog input cache with a specified validity duration.

```c
int load(long msValidity)
```

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

**Parameters:**

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

**Returns:**

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

```
int registerValueCallback( UpdateCallback callback)
```

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

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

Returns the pulse counter value as well as his timer

```c
int resetCounter()
```

**Returns:**

- `API_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
anbutton\rightarrow\text{set\_analog\_Calibration()}
anbutton\rightarrow\text{setAnalog\_Calibration()}
anbutton.set\_analog\_Calibration()

Starts or stops the calibration process.

```
int set\_analog\_Calibration( int newval)
```

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

<table>
<thead>
<tr>
<th>Parameters :</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code> either <code>Y\_ANALOGCALIBRATION\_OFF</code> or <code>Y\_ANALOGCALIBRATION\_ON</code></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 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.
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.
anbutton->set_logicalName()
anbutton->setLogicalName()
anbutton.set_logicalName()

Changes the logical name of the analog input.

```
int set_logicalName( String newVal)
```

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

**Parameters:**
- `newVal` a string corresponding to the logical name of the analog input.

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

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

\texttt{anbutton\rightarrow set\_userData()} \hspace{1cm} \texttt{anbutton\rightarrow setUserData()} \hspace{1cm} \texttt{anbutton.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.

\begin{itemize}
  \item \textbf{Parameters} :
  \begin{itemize}
    \item \textbf{data} any kind of object to be stored
  \end{itemize}
\end{itemize}
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:

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

```
var yoctolib = require('yoctolib');
var YCarbonDioxide = yoctolib.YCarbonDioxide;
```

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

```
#include "yocto_carbondioxide.h"
```

```
#include "yocto_carbondioxide.h"
```

```
uses yocto_carbondioxide;
```

```
yocto_carbondioxide.vb
```

```
yocto_carbondioxide.cs
```

```
import com.yoctopuce.YoctoAPI.YCarbonDioxide;
```

```
from yocto_carbondioxide import *
```

### Global functions

**yFindCarbonDioxide***(func)**

Retrieves a CO2 sensor for a given identifier.

**yFirstCarbonDioxide()**

Starts the enumeration of CO2 sensors currently accessible.

### YCarbonDioxide methods

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

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

**carbondioxide**→**describe**()

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

**carbondioxide**→**get_advertisedValue**()

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

**carbondioxide**→**get_currentRawValue**()

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

- **get_highestValue()**
  - Returns the maximal value observed for the CO2 concentration since the device was started.

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

- **get_logicalName()**
  - Returns the logical name of the CO2 sensor.

- **get_lowestValue()**
  - Returns the minimal value observed for the CO2 concentration since the device was started.

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

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

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

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

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

- **get_unit()**
  - Returns the measuring unit for the CO2 concentration.

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

- **isOnline()**
  - Checks if the CO2 sensor is currently reachable, without raising any error.

- **isOnline_async(callback, context)**
  - Checks if the CO2 sensor is currently reachable, without raising any error (asynchronous version).

- **load(msValidity)**
  - Preloads the CO2 sensor cache with a specified validity duration.

- **loadCalibrationPoints(rawValues, refValues)**
  - Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

- **load_async(msValidity, callback, context)**
  - Preloads the CO2 sensor cache with a specified validity duration (asynchronous version).

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

- **registerTimedReportCallback(callback)**
  - Registers the callback function that is invoked on every periodic timed notification.

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

- **set_highestValue(newval)**
  - Changes the recorded maximal value observed.

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

- **set_logicalName(newval)**
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
Retrieves a CO2 sensor for a given identifier.

**YCarbonDioxide**.FindCarbonDioxide(String func)

The identifier can be specified using several formats:

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

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

**Parameters:**

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

**Returns:**

- a **YCarbonDioxide** object allowing you to drive the CO2 sensor.
YCarbonDioxide.FirstCarbonDioxide()

YCarbonDioxide.FirstCarbonDioxide()

Starts the enumeration of CO2 sensors currently accessible.

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

Returns:
- a pointer to a YCarbonDioxide object, corresponding to the first CO2 sensor currently online, or a null pointer if there are none.
carbondioxide→calibrateFromPoints()  
carbondioxide.calibrateFromPoints()

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

```java
int calibrateFromPoints( ArrayList<Double> rawValues,
                        ArrayList<Double> refValues)
```

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

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

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

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

On failure, throws an exception or returns a negative error code.
Returns a short text that describes unambiguously the instance of the CO2 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 CO2 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 CO2 sensor (no more than 6 characters).</td>
</tr>
</tbody>
</table>

**Returns:**

- a string corresponding to the current value of the CO2 sensor (no more than 6 characters).

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

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.
carbon dioxide \rightarrow \text{get\_current\_Value()}

\text{YCarbonDioxide}
carbon dioxide \rightarrow \text{current\_Value()}
carbon dioxide.\text{get\_current\_Value()}

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

double \text{get\_current\_Value()}

\textbf{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 \text{Y\_CURRENT\_VALUE\_INVALID}.
3. Reference

YCarbonDioxide

carbon dioxide -> get_errorMessage()
carbon dioxide -> errorMessage()
carbon dioxide.get_errorMessage()

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

String get_errorMessage()

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

Returns :
- a string corresponding to the latest error message that occurred while using the CO2 sensor object
YCarbonDioxide
carbon dioxide \rightarrow \text{get\_error\_Type()}
carbon dioxide \rightarrow \text{error\_Type()}
carbon dioxide.get\_error\_Type()

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

```c
int get_errorType()
```

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

**Returns**:

a number corresponding to the code of the latest error that occurred while using the CO2 sensor object
3. Reference

carbon dioxide → get_friendlyName()  \hspace{1cm} YCarbonDioxide
carbon dioxide → friendlyName()
carbon dioxide.get_friendlyName()

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

String get_friendlyName( )

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

Returns:

- a string that uniquely identifies the CO2 sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME.INVALID.
3. Reference

```java
carbon dioxide \rightarrow \text{get\_function\_Descriptor()}
carbon dioxide \rightarrow \text{function\_Descriptor()}
carbon dioxide.\text{get\_function\_Descriptor()}
```

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

```java
String \text{get\_function\_Descriptor()}
```

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\_DESCRIPTION\_INVALID`. 
3. Reference

carbon dioxide \rightarrow \text{get\_functionId()}

carbon dioxide \rightarrow \text{functionId()}

carbon dioxide.get\_functionId()

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

String \text{get\_functionId()}

For example, \text{relay1}

**Returns:**
- a string that identifies the CO2 sensor (ex: \text{relay1})

On failure, throws an exception or returns \text{Y\_FUNCTIONID\_INVALID}. 
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the CO2 sensor in the form <strong>SERIAL.FUNCTIONID</strong>.</td>
</tr>
</tbody>
</table>

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the CO2 sensor (for example `RELAY01-123456.relay1`).

**Returns:**
- a string that uniquely identifies the CO2 sensor (ex: `RELAY01-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
**YCarbonDioxide**

carbon dioxide → get_highestValue()
carbon dioxide → highestValue()
carbon dioxide.get_highestValue()

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**.
YCarbonDioxide

carbon dioxide $\rightarrow$ get_logFrequency()
carbon dioxide $\rightarrow$ logFrequency()
carbon dioxide.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

YCarbonDioxide

carbondioxide \rightarrow \text{get\_logicalName}()  
carbondioxide \rightarrow \text{logicalName}()  
carbondioxide.get\_logicalName()  

Returns the logical name of the CO2 sensor.

\textbf{String get\_logicalName()}  

\textbf{Returns :}  
\begin{itemize}
  \item a string corresponding to the logical name of the CO2 sensor.
\end{itemize}

On failure, throws an exception or returns \texttt{Y\_LOGICALNAME\_INVALID}.
carbon dioxide \rightarrow \text{get\_lowestValue}() \quad \text{YCarbonDioxide}

\text{carbon dioxide} \rightarrow \text{lowestValue}()

\text{carbon dioxide}.\text{get\_lowestValue}()

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

\begin{verbatim}
\text{double get\_lowestValue( )}
\end{verbatim}

\textbf{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 \text{Y\_LOWESTVALUE\_INVALID}.\n
YCarbonDioxide

carbondioxide\rightarrow \text{get\_module}()
carbondioxide\rightarrow \text{module}()
carbondioxide.\text{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 :}

- an instance of \text{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.
3. Reference

carbon dioxide → get_reportFrequency()  
carbon dioxide → 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.
carbon dioxide.get_resolution() → YCarbonDioxide

carbon dioxide.resolution()

carbon dioxide.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.
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.
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.
carbon dioxide \rightarrow \text{isOnline()}

\text{carbon dioxide.isOnline()} \rightarrow \text{YCarbonDioxide}

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

\text{boolean isOnline( )}

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

\textbf{Returns :}
\begin{itemize}
  \item \text{true} if the CO2 sensor can be reached, and \text{false} otherwise
\end{itemize}
Preloads the CO2 sensor cache with a specified validity duration.

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

carbondioxide→loadCalibrationPoints()  
carbon dioxide.loadCalibrationPoints()

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

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**

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

**Returns:**

- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
YCarbonDioxide · nextCarbonDioxide() · YCarbonDioxide

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

YCarbonDioxide nextCarbonDioxide()

**Returns:**

a pointer to a YCarbonDioxide object, corresponding to a CO2 sensor currently online, or a null pointer if there are no more CO2 sensors to enumerate.
碳氧化碳 \( \rightarrow \) \text{registerTimedReportCallback()}

\text{carbonDioxide}.\text{registerTimedReportCallback()} ()

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

\begin{verbatim}
int registerTimedReportCallback( TimedReportCallback 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.

**Parameters:**
- **callback** the callback 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.
carbondioxide.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

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

YCarbonDioxide

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.
carbondioxide->set_logFrequency()
carbondioxide->setLogFrequency()
carbondioxide.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 CO2 sensor.

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

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

**Parameters :**
- `newval` a string corresponding to the logical name of the CO2 sensor.

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

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

| 
| --- |
| **carbon dioxide**→`set_lowestValue()` |
| **carbon dioxide**→`setLowestValue()` |
| **carbon dioxide.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.
### 3. Reference

**carbon dioxide\rightarrow 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>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code></td>
<td>a string corresponding to the timed value notification frequency for this function</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.
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.

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

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

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

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

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

```
cpp
#include "yocto_colorled.h"
```

```
pas
uses yocto_colorled;
```

```
vb
yocto_colorled.vb
```

```
cs
yocto_colorled.cs
```

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

```
py
from yocto_colorpuce.YoctoAPI.YColorLed;
```

---

**Global functions**

- **yFindColorLed(func)**
  - Retrieves an RGB led for a given identifier.

- **yFirstColorLed()**
  - Starts the enumeration of RGB leds currently accessible.

---

**YColorLed methods**

- **colorled→describe()**
  - Returns a short text that describes unambiguously the instance of the RGB led in the form **TYPE(NAME)=SERIAL.FUNCTIONID**.

- **colorled→get_advertisedValue()**
  - Returns the current value of the RGB led (no more than 6 characters).

- **colorled→get_errorMessage()**
  - Returns the error message of the latest error with the RGB led.

- **colorled→get_errorType()**
  - Returns the numerical error code of the latest error with the RGB led.

- **colorled→get_friendlyName()**
  - Returns a global identifier of the RGB led in the format **MODULE_NAME.FUNCTION_NAME**.

- **colorled→get_functionDescriptor()**
  - Returns a unique identifier of type **YFUN_DESCR** corresponding to the function.

- **colorled→get_functionId()**
  - Returns the hardware identifier of the RGB led, without reference to the module.

- **colorled→get_hardwareId()**
  - Returns the unique hardware identifier of the RGB led in the form **SERIAL.FUNCTIONID**.

- **colorled→get_hslColor()**
  - Returns the current HSL color of the led.

- **colorled→get_logicalName()**
  - Returns the logical name of the RGB led.
3. Reference

<table>
<thead>
<tr>
<th>Method Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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_rgbColor()</code></td>
<td>Returns the current RGB color of the led.</td>
</tr>
<tr>
<td><code>get_rgbColorAtPowerOn()</code></td>
<td>Returns the configured color to be displayed when the module is turned on.</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>hslMove(hsl_target, ms_duration)</code></td>
<td>Performs a smooth transition in the HSL color space between the current color and a target color.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the RGB led is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the RGB led is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the RGB led cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the RGB led cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextColorLed()</code></td>
<td>Continues the enumeration of RGB leds started using <code>yFirstColorLed()</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>rgbMove(rgb_target, ms_duration)</code></td>
<td>Performs a smooth transition in the RGB color space between the current color and a target color.</td>
</tr>
<tr>
<td><code>set_hslColor(newval)</code></td>
<td>Changes the current color of the led, using a color HSL.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td>Changes the logical name of the RGB led.</td>
</tr>
<tr>
<td><code>set_rgbColor(newval)</code></td>
<td>Changes the current color of the led, using a RGB color.</td>
</tr>
<tr>
<td><code>set_rgbColorAtPowerOn(newval)</code></td>
<td>Changes the color that the led will display by default when the module is turned on.</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>
YColorLed.FindColorLed() YColorLed

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.
Starts the enumeration of RGB leds currently accessible.

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 \( \text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID} \).

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 RGB led (ex: \text{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1})
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>colorled.get_advertisedValue()</code></td>
<td>Returns the current value of the RGB led (no more than 6 characters).</td>
</tr>
</tbody>
</table>

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

String get_errorMessage()

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

Returns:

- a string corresponding to the latest error message that occurred while using the RGB led object
Returns the numerical error code of the latest error with the RGB led.

```c
int get_errorType()
```

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

**Returns:**

a number corresponding to the code of the latest error that occurred while using the RGB led object
>Returns a global identifier of the RGB led in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName( )

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

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

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

YColorLed

colorled→get_functionDescriptor()
colorled→functionDescriptor()
colorled.get_functionDescriptor()

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

**String get_functionDescriptor( )**

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

**Returns :**

an identifier of type `YFUN_DESCR`.

If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.
Returns the hardware identifier of the RGB led, without reference to the module.

```java
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`. 
### 3. Reference

<table>
<thead>
<tr>
<th>colorled.get_hardwareId()</th>
<th>YColorLed</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorled.hardwareId()</td>
<td></td>
</tr>
<tr>
<td>colorled.get_hardwareId()</td>
<td></td>
</tr>
</tbody>
</table>

Returns the unique hardware identifier of the RGB led 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 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`. 
**get_hslColor()**

Returns the current HSL color of the led.

```cpp
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.

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.
YColorLed

**colorled.get_module()**

**colorled.module()**

**colorled.get_module()**

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

YModule `get_module()`

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

**Returns:**

an instance of `YModule`
Returns the current RGB color of the led.

int get_rgbColor()

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

On failure, throws an exception or returns Y_RGBCOLOR_INVALID.
Returns the configured color to be displayed when the module is turned on.

**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`. 
3. Reference

YColorLed

colorled→get_userData()
colorled→userData()colorled.get_userData()

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

Object get_userData()

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

Returns :
the object stored previously by the caller.
colorled→hslMove() colorled.hslMove()

Performs 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()**

Preloads the RGB led cache with a specified validity duration.

```c
int load( long msValidity)
```

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

**Parameters :**

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

**Returns :**

- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
colorled→nextColorLed()  
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.
3. Reference

**colorled.registerValueCallback()**

colorled.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

- **callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
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.
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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>colorled.set_rgbColorAtPowerOn()</code></td>
<td>Changes the color that the led will display by default when the module is turned on.</td>
</tr>
<tr>
<td><code>int set_rgbColorAtPowerOn( int newval)</code></td>
<td>This color will be displayed as soon as the module is powered on. Remember to call the <code>saveToFlash()</code> method of the module if the change should be kept.</td>
</tr>
</tbody>
</table>

**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.

```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
3.7. 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
#include "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.
### 3. Reference

**compass→get_highestValue()**  
Returns the maximal value observed for the relative bearing since the device was started.

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

**compass→get_logicalName()**  
Returns the logical name of the compass.

**compass→get_lowestValue()**  
Returns the minimal value observed for the relative bearing since the device was started.

**compass→get_magneticHeading()**  
Returns the magnetic heading, regardless of the configured bearing.

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

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

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

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

**compass→get_resolution()**  
Returns the resolution of the measured values.

**compass→get_unit()**  
Returns the measuring unit for the relative bearing.

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

**compass→isOnline()**  
Checks if the compass is currently reachable, without raising any error.

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

**compass→load(msValidity)**  
Preloads the compass cache with a specified validity duration.

**compass→loadCalibrationPoints(rawValues, refValues)**  
Retrieves error correction data points previously entered using the method calibrateFromPoints.

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

**compass→nextCompass()**  
Continues the enumeration of compasses started using yFirstCompass().

**compass→registerTimedReportCallback(callback)**  
Registers the callback function that is invoked on every periodic timed notification.

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

**compass→set_highestValue(newval)**  
Changes the recorded maximal value observed.

**compass→set_logFrequency(newval)**  
Changes the datalogger recording frequency for this function.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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()**

`yFindCompass() YCompass.FindCompass()`

Retrieves a compass for a given identifier.

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

The identifier can be specified using several formats:

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

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

**Parameters :**

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

**Returns :**

- a `YCompass` object allowing you to drive the compass.
YCompass.FirstCompass()

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.
**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(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

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

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

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

compass.describe() → YCompass

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

String describe()

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

Returns:

a string that describes the compass (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
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()  \textit{YCompass}
compass→currentRawValue()
compass.get_currentRawValue()

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

\begin{verbatim}
double get_currentRawValue()
\end{verbatim}

\textbf{Returns :}
\begin{itemize}
  \item a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number
\end{itemize}

On failure, throws an exception or returns \texttt{Y_CURRENTRAWVALUE_INVALID}.
**compass → get_currentValue()**  
**compass → currentValue()**  
**compass.get_currentValue()**

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\rightarrow get\_errorMessage()

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

**String get\_errorMessage()**

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

**Returns:**

- a string corresponding to the latest error message that occurred while using the compass object
Returns the numerical error code of the latest error with the compass.

```c
int get_errorType()
```

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

**Returns:**

A number corresponding to the code of the latest error that occurred while using the compass object.
YCompass
compass→get_friendlyName()
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.
compass → get_functionDescriptor()
compass → functionDescriptor()
compass.get_functionDescriptor()

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

String get_functionDescriptor( )

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

**Returns:**

- an identifier of type YFUN_DESCR.

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

compass.get_hardwareId()
compass.hardwarId()
compass.get_hardwarId()

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

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

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

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

YCompass

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

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

double get_highestValue()

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
compass→get_logFrequency()
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.
3. Reference

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

Returns the logical name of the compass.

String get_logicalName() Returns:

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

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
3. Reference

YCompass

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

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

double get_lowestValue()

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
3. Reference

YCompass

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

Returns the magnetic heading, regardless of the configured bearing.

\[
\text{double } \text{get}_\text{magneticHeading}( )
\]

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

On failure, throws an exception or returns \text{Y\_MAGNETIC\_HEADING\_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
3. Reference

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.
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`. 
### compass → get_unit()

**compass → unit()**

*compass.get_unit()*

Returns the measuring unit for the relative bearing.

**String get_unit()**

**Returns:**

- a string corresponding to the measuring unit for the relative bearing

On failure, throws an exception or returns **Y_UNIT_INVALID**.
3. Reference

YCompass

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

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

Object get_userData()

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

Returns:
the object stored previously by the caller.
YCompass

compass→isOnline() \(\text{compass.isOnline()}\)

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

boolean isOnline()

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

**Returns**:
true if the compass can be reached, and false otherwise
Preloads the compass 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.
### loadCalibrationPoints

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

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rawValues</code></td>
<td>array of floating point numbers, that will be filled by the function with</td>
</tr>
<tr>
<td></td>
<td>the raw sensor values for the correction points.</td>
</tr>
<tr>
<td><code>refValues</code></td>
<td>array of floating point numbers, that will be filled by the function with</td>
</tr>
<tr>
<td></td>
<td>the desired 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>
<tr>
<td></td>
<td>On failure, throws an exception or returns a negative error code.</td>
</tr>
</tbody>
</table>
### compass.nextCompass() \( \text{compass.nextCompass()} \)

Continues the enumeration of compasses started using \( \text{yFirstCompass()} \).

<table>
<thead>
<tr>
<th><strong>Returns</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a pointer to a ( \text{YCompass} ) object, corresponding to a compass currently online, or a null pointer if there are no more compasses to enumerate.</td>
<td></td>
</tr>
</tbody>
</table>
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.
YCompass
compass.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**
- **callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
### compass \(\rightarrow\) set\_highestValue() YCompass
compass \(\rightarrow\) setHighestValue()
compass.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.

```java
int set_logFrequency(String newval)
```

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

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

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

On failure, throws an exception or returns a negative error code.
compass.set_logicalName()
changes the logical name of the compass.

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

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

**Parameters:**
- `newval` a string corresponding to the logical name of the compass.

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

On failure, throws an exception or returns a negative error code.
compass→set_lowestValue()  \hspace{1cm} \text{YCompass}
compass→setLowestValue()
compass.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.

```java
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.

`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;
```

```python
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current.get_highestValue()</code></td>
<td>Returns the maximal value observed for the current since the device was started.</td>
</tr>
<tr>
<td><code>current.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>current.get_logicalName()</code></td>
<td>Returns the logical name of the current sensor.</td>
</tr>
<tr>
<td><code>current.get_lowestValue()</code></td>
<td>Returns the minimal value observed for the current since the device was started.</td>
</tr>
<tr>
<td><code>current.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>current.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>current.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>current.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>current.get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>current.get_unit()</code></td>
<td>Returns the measuring unit for the current.</td>
</tr>
<tr>
<td><code>current.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>current.isOnline()</code></td>
<td>Checks if the current sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>current.isOnline_async(callback, context)</code></td>
<td>Checks if the current sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>current.load(msValidity)</code></td>
<td>Preloads the current sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>current.loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>current.load_async(msValidity, callback, context)</code></td>
<td>Preloads the current sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>current.nextCurrent()</code></td>
<td>Continues the enumeration of current sensors started using <code>yFirstCurrent()</code>.</td>
</tr>
<tr>
<td><code>current.registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>current.registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>current.set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>current.set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>current.set_logicalName(newval)</code></td>
<td>Changes the logical name of the current sensor.</td>
</tr>
</tbody>
</table>
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
<td></td>
</tr>
<tr>
<td><code>current.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
<td></td>
</tr>
<tr>
<td><code>current.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
<td></td>
</tr>
<tr>
<td><code>current.set_userData(data)</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
<td></td>
</tr>
<tr>
<td><code>current.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>
<td></td>
</tr>
</tbody>
</table>
YCurrent.FindCurrent()
yFindCurrent() YCurrent.FindCurrent()

Retrieves a current sensor for a given identifier.

YCurrent FindCurrent( String func)

The identifier can be specified using several formats:

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

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

Parameters :
- func a string that uniquely characterizes the current sensor

Returns :
a YCurrent object allowing you to drive the current sensor.
YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()

YCurrent.FirstCurrent()
current → calibrateFromPoints()  

```
current.calibrateFromPoints()
```

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

```
int calibrateFromPoints( ArrayList<Double> rawValues,
                        ArrayList<Double> refValues)
```

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

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

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

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

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

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

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

String get_advertisedValue()  

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

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

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 \texttt{Y\_CURRENTRAWVALUE\_INVALID}.
current \rightarrow \text{get\_current\_Value()}

current \rightarrow \text{current\_Value()}

\text{current.get\_current\_Value()}

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

\text{double get\_current\_Value()}

\textbf{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 \text{Y\_CURRENT\_VALUE\_INVALID}. 
3. Reference

`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
### YCurrent

#### current→get_errorType()

`current->errorType()`

```c
int get_errorType()
```

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

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

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

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

String get_functionDescriptor( )

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

Returns:
  an identifier of type YFUN_DESCR.

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

`current.get_functionId()`

**YCurrent**

`current.functionId()`

`current.get_functionId()`

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

`String get_functionId()`

For example `relay1`

**Returns**:

- A string that identifies the current sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
current→get_hardwareId()
current→hardwareId() current.get_hardwareId()

Returns the unique hardware identifier of the current sensor in the form SERIAL_FUNCTIONID.

String get_hardwareId()

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

Returns:

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
Reference

current\rightarrow\text{get\_highest\_Value()}

\text{current\_get\_highest\_Value()}

YCurrent

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

double \text{get\_highest\_Value()}

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 \text{Y\_HIGHEST\_VALUE\_INVALID}.
current→get_logFrequency()
current→logFrequency()
current.get_logFrequency()

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

String get_logFrequency( )

Returns:

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.
current→get_logicalName() YCurrent
current→logicalName()
current.get_logicalName() Returns the logical name of the current sensor.

String get_logicalName( )

Returns:
a string corresponding to the logical name of the current sensor.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
<table>
<thead>
<tr>
<th>current→get_lowestValue()</th>
<th>YCurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>current→lowestValue()</td>
<td></td>
</tr>
<tr>
<td>current.get_lowestValue()</td>
<td></td>
</tr>
</tbody>
</table>

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

```java
double get_lowestValue() 
```

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
current \rightarrow \text{get\_module()}

current \rightarrow \text{module()}

current.get\_module()

\textbf{YCurrent}

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

\begin{itemize}
\item \texttt{YModule get\_module()}
\end{itemize}

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

\begin{itemize}
\item \textbf{Returns :}
\item an instance of \texttt{YModule}
\end{itemize}
current→get_recordedData()
current→recordedData()
current.get_recordedData()

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

YDataSet get_recordedData( long startTime, long endTime)

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

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

Parameters :
- **startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any 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\rightarrow get\_resolution()

YCurrent

current\rightarrow resolution()\texttt{current}\texttt{.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 \texttt{Y\_RESOLUTION\_INVALID}.
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.
Checks if the current sensor is currently reachable, without raising any error.

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

Returns:
- true if the current sensor can be reached, and false otherwise
current.load() current.load()

Preloads the current sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

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

**Parameters :**

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

**Returns :**

- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
current\rightarrow & \text{loadCalibrationPoints()}
\end{align*}

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

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

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

Continues the enumeration of current sensors started using \texttt{yFirstCurrent()}. 

\texttt{YCurrent nextCurrent()}

\textbf{Returns:}

- a pointer to a \texttt{YCurrent} object, corresponding to a current sensor currently online, or a \texttt{null} pointer if there are no more current sensors to enumerate.
3. Reference

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

**int registerTimedReportCallback(TimedReportCallback callback)**

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

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.
current->registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

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

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.

```java
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.

### 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\rightarrow \text{set\_lowest\_Value}() \\
\texttt{current\rightarrow setLowestValue()} \\
\texttt{current.set\_lowest\_Value()} \\

Changes the recorded minimal value observed.

\begin{Verbatim}
\textbf{int set\_lowest\_Value( double newval)}
\end{Verbatim}

\begin{description}
\item[Parameters:] \texttt{newval} a floating point number corresponding to the recorded minimal value observed
\item[Returns:] \texttt{YAPI\_SUCCESS} if the call succeeds.
\end{description}

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.

**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.

`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"
```

```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()**
- Returns true if the data logger is synchronised with the localization beacon.

**datalogger→get_currentRunIndex()**
- Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

**datalogger→get_dataSets()**
- Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

**datalogger→get_dataStreams(v)**
- Builds a list of all data streams held 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>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.</td>
</tr>
<tr>
<td><code>get_functionId()</code></td>
<td>Returns the hardware identifier of the data logger, without reference to the module.</td>
</tr>
<tr>
<td><code>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>get_logicalName()</code></td>
<td>Returns the logical name of the data logger.</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()</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_recording()</code></td>
<td>Returns the current activation state of the data logger.</td>
</tr>
<tr>
<td><code>get_timeUTC()</code></td>
<td>Returns the Unix timestamp for current UTC time, if known.</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 data logger is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async()</code></td>
<td>Checks if the data logger is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the data logger cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>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>nextDataLogger()</code></td>
<td>Continues the enumeration of data loggers started using <code>yFirstDataLogger()</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_autoStart(newval)</code></td>
<td>Changes the default activation state of the data logger on power up.</td>
</tr>
<tr>
<td><code>set_beaconDriven(newval)</code></td>
<td>Changes the type of synchronisation of the data logger.</td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
<td>Changes the logical name of the data logger.</td>
</tr>
<tr>
<td><code>set_recording(newval)</code></td>
<td>Changes the activation state of the data logger to start/stop recording data.</td>
</tr>
<tr>
<td><code>set_timeUTC(newval)</code></td>
<td>Changes the current UTC time reference used for recorded data.</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></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()

YDataLogger.FindDataLogger()

Retrieves a data logger for a given identifier.

YDataLogger FindDataLogger( String func)

The identifier can be specified using several formats:

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

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

Parameters :

- func a string that uniquely characterizes the data logger

Returns :

- a YDataLogger object allowing you to drive the data logger.
YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()

YDataLogger.FirstDataLogger()
YDataLogger.describe()  

Returns a short text that describes unambiguously the instance of the data logger 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(BadCustomerName.relay1)=unresolved if the module
has not yet been connected. This method does not trigger any USB or TCP transaction and can
therefore be used in a debugger.

Returns:
a string that describes the data logger (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
YDataLogger

YDataLogger

datalogger→forgetAllDataStreams()  YDataLogger

datalogger.forgetAllDataStreams()

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

int forgetAllDataStreams( )

This method also resets the current run index to zero.

Returns :

YAPI_SUCCESS if the call succeeds.

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

YDataLogger

- `datalogger→get_advertisedValue()`
- `advertisedValue()`
- `datalogger.get_advertisedValue()`

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

**String get_advertisedValue( )**

**Returns:**

- a string corresponding to the current value of the data logger (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
datalogger\rightarrow get\_autoStart() \hspace{1cm} YDataLogger
datalogger\rightarrow autoStart() \hspace{1cm} datalogger.get\_autoStart()

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

int get\_autoStart()

**Returns:**

either Y AUTOSTART OFF or Y AUTOSTART ON, according to the default activation state of the data logger on power up

On failure, throws an exception or returns Y AUTOSTART INVALID.
Return true if the data logger is synchronised with the localization beacon.

```c
int get_beaconDriven() {
    // Implementation details...
    return Y_BEACONDRIVEN_OFF or Y_BEACONDRIVEN_ON;
}
```

On failure, throws an exception or returns Y_BEACONDRIVEN_INVALID.
datalogger→get_currentRunIndex()
datalogger→currentRunIndex()
datalogger.get_currentRunIndex()

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

int get_currentRunIndex( )

Returns:

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

On failure, throws an exception or returns Y_CURRENTRUNINDEX_INVALID.
YDataLogger

YDataLogger

dataLogger→get_dataSets()
dataLogger→dataSets()

dataLogger.get_dataSets()

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

ArrayList<YDataSet> get_dataSets()

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

Returns:

- a list of YDataSet object.

On failure, throws an exception or returns an empty list.
YDataLogger

datalogger→get_dataStreams()
datalogger→dataStreams()
datalogger.get_dataStreams ()

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

```java
int get_dataStreams( ArrayList<YDataStream> v)
```

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

**Parameters:**

- `v` an array of YDataStream objects to be filled in

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Returns the error message of the latest error with the data logger.

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

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

Returns:
- a string corresponding to the latest error message that occurred while using the data logger object
Returns the numerical error code of the latest error with the data logger.

```c
int get_errorType()
```

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

**Returns**:

A number corresponding to the code of the latest error that occurred while using the data logger object.
3. Reference

datalogger→get_friendlyName()
datalogger→friendlyName()
datalogger.get_friendlyName()

YDataLogger

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

\texttt{String 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.

\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 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`. 
Returns the unique hardware identifier of the data logger 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 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`.
<table>
<thead>
<tr>
<th>YDataLogger</th>
</tr>
</thead>
<tbody>
<tr>
<td>datalogger→get_logicalName()</td>
</tr>
<tr>
<td>datalogger→logicalName()</td>
</tr>
<tr>
<td>datalogger.get_logicalName()</td>
</tr>
</tbody>
</table>

| String get_logicalName() |

- Returns the logical name of the data logger.

  Returns:
  a string corresponding to the logical name of the data logger.

  On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datalogger.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>YModule.get_module()</code></td>
<td>If the function cannot be located on any module, the returned instance of <code>YModule</code> is not shown as online.</td>
</tr>
</tbody>
</table>

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

datalogger→get_recording()
datalogger→recording()
datalogger.get_recording()

Returns the current activation state of the data logger.

```c
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`. 
Returns the Unix timestamp for current UTC time, if known.

Returns:
- an integer corresponding to the Unix timestamp for current UTC time, if known
- On failure, throws an exception or returns \texttt{Y\_TIMEUTC\_INVALID}.
YDataLogger

3. Reference

datalogger→get_userData()
datalogger→userData()
datalogger.get_userData()

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

Object get_userData()

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

Returns:
the object stored previously by the caller.
datalogger→isOnline() YDataLogger

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

boolean isOnline()

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

Returns:
true if the data logger can be reached, and false otherwise
**datalogger→load()**

Preloads the data logger cache with a specified validity duration.

```c
int load( long msValidity)
```

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

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

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

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

**YDataLogger `nextDataLogger()`**

**Returns**: a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a null pointer if there are no more data loggers to enumerate.
Registrates the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
Changes the default activation state of the data logger on power up.

```c
int set_autoStart(int newval)
```

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

**Parameters:**
- `newval` either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up.

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

```plaintext
dllogger→set_beaconDriven()
dllogger→setBeaconDriven()
dllogger.set_beaconDriven()
```

Changes the type of synchronisation of the data logger.

```plaintext
int set_beaconDriven( int newval)
```

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

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

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

On failure, throws an exception or returns a negative error code.
Changes the logical name of the data logger.

```java
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.
YDataLogger

datalogger→set_recording()
datalogger→setRecording()
datalogger.set_recording()

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.

**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.
3. Reference

datalogger→set_userData()
datalogger→setUserData()
datalogger.set_userData() YDataLogger

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
3.10. 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"
```

```m
#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 *
```

### YDataRun methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datarun.get_averageValue(measureName, pos)</code></td>
<td>Returns the average value of the measure observed at the specified time period.</td>
</tr>
<tr>
<td><code>datarun.get_duration()</code></td>
<td>Returns the duration (in seconds) of the data run.</td>
</tr>
<tr>
<td><code>datarun.get_maxValue(measureName, pos)</code></td>
<td>Returns the maximal value of the measure observed at the specified time period.</td>
</tr>
<tr>
<td><code>datarun.get_measureNames()</code></td>
<td>Returns the names of the measures recorded by the data logger.</td>
</tr>
<tr>
<td><code>datarun.get_minValue(measureName, pos)</code></td>
<td>Returns the minimal value of the measure observed at the specified time period.</td>
</tr>
<tr>
<td><code>datarun.get_startTimeUTC()</code></td>
<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>
<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>
<td>Returns the number of seconds covered by each value in this run.</td>
</tr>
<tr>
<td><code>datarun.set_valueInterval(valueInterval)</code></td>
<td>Changes the number of seconds covered by each value in this run.</td>
</tr>
</tbody>
</table>
Returns the average value of the measure observed at the specified time period.

```java
double get_averageValue(String measureName, int pos)
```

Parameters:

- `measureName`: the name of the desired measure (one of the names returned by `get_measureNames`)
- `pos`: the position index, between 0 and the value returned by `get_valueCount`

Returns:

- a floating point number (the average value)

On failure, throws an exception or returns `Y_AVERAGEVALUE_INVALID`. 
datarun→get_duration()

datarun→duration() datarun.get_duration()

Returns the duration (in seconds) of the data run.

long get_duration()

datarun→get_duration()

datarun→duration() datarun.get_duration()

Returns the duration (in seconds) of the data run.

js

function get_duration()

nodejs

function get_duration()

php

function get_duration()

java

long get_duration()

py

def get_duration()

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

Returns:

an unsigned number corresponding to the number of seconds between the beginning of the run (when the module was powered up) and the last recorded measure.
Returns the maximal value of the measure observed at the specified time period.

```
datarun->get_maxValue()
datarun->maxValue() datarun.get_maxValue()
```

```java
double get_maxValue(String measureName, int pos)
```

```javascript
function get_maxValue(measureName, pos)
```

```php
function get_maxValue($measureName, $pos)
```

```python
def get_maxValue(measureName, pos)
```

**Parameters:**

- `measureName`: the name of the desired measure (one of the names returned by `get_measureNames`)
- `pos`: the position index, between 0 and the value returned by `get_valueCount`

**Returns:**

- a floating point number (the maximal value)

On failure, throws an exception or returns `Y_MAXVALUE_INVALID`. 
YDataRun

dataRun → get_measureNames()
dataRun → measureNames()
dataRun.get_measureNames()

Returns the names of the measures recorded by the data logger.

ArrayList<String> get_measureNames()  

dataRun → get_measureNames()
dataRun → measureNames()  

dataRun.get_measureNames()

Returns the names of the measures recorded by the data logger.

Java
ArrayList<String> get_measureNames()

In most case, the measure names match the hardware identifier of the sensor that produced the data.

Returns:

a list of strings (the measure names) On failure, throws an exception or returns an empty array.
3. Reference

YDataRun

datarun\rightarrow get\_minValue()  
datarun\rightarrow minValue()  
datarun.get_minValue()  

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

```java
double get_minValue(String measureName, int pos)
```

```javascript
function get_minValue( measureName, pos)
```

```nodejs
function get_minValue( measureName, pos)
```

```php
function get_minValue( $measureName, $pos)
```

```java
double get_minValue( String measureName, int pos)
```

```python
def get_minValue( measureName, pos)
```

Parameters :

- **measureName** the name of the desired measure (one of the names returned by `get_measureNames`)
- **pos** the position index, between 0 and the value returned by `get_valueCount`

Returns :

- a floating point number (the minimal value)

On failure, throws an exception or returns Y\_MINVALUE\_INVALID.
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).
Returns the number of values accessible in this run, given the selected data samples interval.

```javascript
function get_valueCount() {
}
```

```nodejs
function get_valueCount() {
}
```

```php
function get_valueCount() {
}
```

```java
int get_valueCount() {
}
```

```python
def get_valueCount() {
}
```

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

**Returns:**

A signed number corresponding to the run duration divided by the samples interval.
YDataRun

datarun→get_valueInterval()
datarun→valueInterval()
datarun.get_valueInterval()

Returns the number of seconds covered by each value in this run.

int get_valueInterval( )

datarun→get_valueInterval()
datarun→valueInterval() datarun.get_valueInterval()

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Returns:

an unsigned number corresponding to a number of seconds covered by each data sample in the Run.
Changes the number of seconds covered by each value in this run.

```java
void set_valueInterval( int valueInterval)
```

**Parameters:**

- `valueInterval` an integer number of seconds.

**Returns:**

nothing

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.
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 that data will be effectively loaded from the dataLogger.

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

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

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

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

```nodejs
var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
```

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

```cpp
#include "yocto_api.h"
```

```m
#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 *
```

### YDataSet methods

- **`dataset→get_endTimeUTC()`**
  Returns the end time of the dataset, relative to the Jan 1, 1970.

- **`dataset→get_functionId()`**
  Returns the hardware identifier of the function that performed the measure, without reference to the module.

- **`dataset→get_hardwareId()`**
  Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

- **`dataset→get_measures()`**
  Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

- **`dataset→get_preview()`**
  Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

- **`dataset→get_progress()`**
  Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

- **`dataset→get_startTimeUTC()`**
  Returns the start time of the dataset, relative to the Jan 1, 1970.

- **`dataset→get_summary()`**
  Returns an YMeasure object which summarizes the whole DataSet.

- **`dataset→get_unit()`**
  Returns the measuring unit for the measured value.
3. Reference

**dataset→loadMore()**

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

**dataset→loadMore_async(callback, context)**

Loads the next block of measures from the dataLogger asynchronously.
YDataSet

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

When the YDataSet is created, the end time is the value passed in parameter to the \texttt{get\_dataSet()} function. After the very first call to \texttt{loadMore()}, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

\begin{itemize}
\item \textbf{Returns :}
\item \texttt{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).}
\end{itemize}
Returns the hardware identifier of the function that performed the measure, without reference to the module.

String `get_functionId()`

For example `temperature1`.

**Returns:**
- a string that identifies the function (ex: `temperature1`)
**dataset.get_hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

**String get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `THRMCPL1-123456.temperature1`).

**Returns:**

a string that uniquely identifies the function (ex: `THRMCPL1-123456.temperature1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
dataset→get_measures()  YDataSet
dataset→measures()  dataset.get_measures()  

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

`ArrayList<YMeasure> get_measures()`

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns:

- a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.
Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

`ArrayList<YMeasure> get_preview()`

Each item includes:
- the start of a time interval
- the end of a time interval
- the minimal value observed during the time interval
- the average value observed during the time interval
- the maximal value observed during the time interval

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

**Returns:**
- a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.
3. Reference

YDataSet

dataset→get_progress()
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()

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

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).
3. Reference

**dataset→get_summary()**

YDataSet

dataset→summary()dataset.get_summary()

Returns an YMeasure object which summarizes the whole DataSet.

YMeasure get_summary()

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

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

**Returns:**
- an YMeasure object
```yaml
dataset.get_unit()
```

Returns the measuring unit for the measured value.

```java
String get_unit()
```

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

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

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

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:

```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"
```
```m
#import "yocto_api.h"
```
```pas
uses yocto_api;
```
```vb
yocto_api.vb
```
```cs
from yocto_api import *
```
```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.

```java
double get_averageValue()
```

If the device uses a firmware older than version 13000, this method will always return `Y_DATA_INVALID`.

**Returns:**
- A floating-point number corresponding to the average value, or `Y_DATA_INVALID` if the stream is not yet complete (still recording).

On failure, throws an exception or returns `Y_DATA_INVALID`. 
Returns the number of data columns present in this stream.

\[
\text{int } \text{get\_columnCount()}
\]

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

`ArrayList<String> get_columnNames()`

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the data Logger 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.
YDataStream

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.
3. Reference

YDataStream
datastream → get_dataRows()
datastream → dataRows()
datastream.get_dataRows()

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

```
ArrayList<ArrayList<Double>> get_dataRows()
```

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

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

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

On failure, throws an exception or returns an empty array.
YDataStream

datastream
rightarrow
dataSamplesIntervalMs()

YDataStream
datastream
rightarrow
dataSamplesIntervalMs()

Returns the number of milliseconds between two consecutive rows of this data stream.

int get_dataSamplesIntervalMs()

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

Returns:
an unsigned number corresponding to a number of milliseconds.
Returns the approximate duration of this stream, in seconds.

<table>
<thead>
<tr>
<th>int get_duration()</th>
</tr>
</thead>
</table>

Returns:
the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.
3. Reference

YDataStream

datastream→get_maxValue()  
datastream→maxValue()  
datastream.get_maxValue()

Returns the largest measure observed within this stream.

Returns:

- a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.
Returns the smallest measure observed within this stream.

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.

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()
```

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

**Returns:**

an unsigned number corresponding to the run index.
YDataStream

datastream\rightarrow get\_startTime()
datastream\rightarrow startTime()
datastream.get\_startTime()

Returns the relative start time of the data stream, measured in seconds.

```c
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.

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:

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

```javascript
var yoctolib = require('yoctolib');
var YDigitalIO = yoctolib.YDigitalIO;
```

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

```cpp
#include "yocto_digitalio.h"
```

```python
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`.
<p>| digitalio\rightarrow get_functionDescriptor() | Returns a unique identifier of type YFUN_DESCR corresponding to the function. |
| digitalio\rightarrow get_functionId() | Returns the hardware identifier of the digital IO port, without reference to the module. |
| digitalio\rightarrow get_hardwareId() | Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID. |
| digitalio\rightarrow get_logicalName() | Returns the logical name of the digital IO port. |
| digitalio\rightarrow get_module() | Gets the YModule object for the device on which the function is located. |
| digitalio\rightarrow get_module_async(callback, context) | Gets the YModule object for the device on which the function is located (asynchronous version). |
| digitalio\rightarrow get_outputVoltage() | Returns the voltage source used to drive output bits. |
| digitalio\rightarrow get_portDirection() | Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output. |
| digitalio\rightarrow get_portOpenDrain() | Returns the electrical interface for each bit of the port. |
| digitalio\rightarrow get_portPolarity() | Returns the polarity of all the bits of the port. |
| digitalio\rightarrow get_portSize() | Returns the number of bits implemented in the I/O port. |
| digitalio\rightarrow get_portState() | Returns the digital IO port state: bit 0 represents input 0, and so on. |
| digitalio\rightarrow get_userData() | Returns the value of the userData attribute, as previously stored using method set_userData. |
| digitalio\rightarrow isOnline() | Checks if the digital IO port is currently reachable, without raising any error. |
| digitalio\rightarrow isOnline_async(callback, context) | Checks if the digital IO port is currently reachable, without raising any error (asynchronous version). |
| digitalio\rightarrow load(msValidity) | Preloads the digital IO port cache with a specified validity duration. |
| digitalio\rightarrow load_async(msValidity, callback, context) | Preloads the digital IO port cache with a specified validity duration (asynchronous version). |
| digitalio\rightarrow nextDigitalIO() | Continues the enumeration of digital IO ports started using yFirstDigitalIO(). |
| digitalio\rightarrow pulse(bitno, ms_duration) | Triggers a pulse on a single bit for a specified duration. |
| digitalio\rightarrow registerValueCallback(callback) | Registers the callback function that is invoked on every change of advertised value. |
| digitalio\rightarrow set_bitDirection(bitno, bitdirection) | Changes the direction of a single bit from the I/O port. |
| digitalio\rightarrow set_bitOpenDrain(bitno, opendrain) | Changes the electrical interface of a single bit from the I/O port. |
| digitalio\rightarrow set_bitPolarity(bitno, bitpolarity) |</p>
<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>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 IO 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>
YDigitalIO.FindDigitalIO()

Retrieves a digital IO port for a given identifier.

YDigitalIO.FindDigitalIO( String func)

The identifier can be specified using several formats:

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

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

Parameters :

    func a string that uniquely characterizes the digital IO port

Returns :

    a YDigitalIO object allowing you to drive the digital IO port.
YDigitalIO.FirstDigitalIO()

YDigitalIO.FirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

YDigitalIO.FirstDigitalIO() method

Use the method YDigitalIO.nextDigitalIO() to iterate on next digital IO ports.

Returns:
The YDigitalIO object representing the first digital IO port currently online, or a null pointer if none exist.
digitalio.delayedPulse()

digitalio.delayedPulse()

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

int delayedPulse( int bitno, int ms_delay, int ms_duration)

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

- **bitno** the bit number; lowest bit has index 0
- **ms_delay** waiting time before the pulse, in milliseconds
- **ms_duration** desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

- **YAPI_SUCCESS** if the call succeeds.

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

**digitalio** → **describe()**

YDigitalIO

Returns a short text that describes unambiguously the instance of the digital IO port in the form

\[ \text{TYPE}(\text{NAME}) = \text{SERIAL}.\text{FUNCTIONID}. \]

**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 \( \text{Relay(MyCustomName.relay1)} = \text{RELAYLO1-123456.relay1} \) 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 digital IO port (ex: Relay(MyCustomName.relay1) = RELAYLO1-123456.relay1)
Returns the current value of the digital IO port (no more than 6 characters).

**String get_advertisedValue()**

- **Returns:**
  - a string corresponding to the current value of the digital IO port (no more than 6 characters).

- On failure, throws an exception or returns **Y_ADVERTISEDVALUE_INVALID**.
digitalio→get_bitDirection()       YDigitalIO
digitalio→bitDirection()          digitalio.get_bitDirection()

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

int get_bitDirection( int bitno)

Parameters:
- bitno the bit number; lowest bit has index 0

Returns:
- YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
digitalio→get_bitOpenDrain()
digitalio→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 a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

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

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() YDigitalIO

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

String get_errorMessage( )

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

**Returns** :

a string corresponding to the latest error message that occurred while using the digital IO port object
Returns the numerical error code of the latest error with the digital IO port.

```c
int get_errorType()
```

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

Returns:
- a number corresponding to the code of the latest error that occurred while using the digital IO port object
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.

String get_functionDescriptor( )

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

**Returns:**

an identifier of type YFUN_DESCR.

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

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 RELAYLO1-123456.relay1).

Returns :
  a string that uniquely identifies the digital IO port (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
digitalio→get_logicalName()  YDigitalIO
digitalio→logicalName()  digitalio.get_logicalName()

Returns the logical name of the digital IO port.

String get_logicalName( )

Returns:
  a string corresponding to the logical name of the digital IO port.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
YDigitalIO
digitalio→get_module()
digitalio→module()digitalio.get_module()

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

YModule get_module()

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

Returns:
an instance of YModule
Returns the voltage source used to drive output bits.

```c
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.
}
```
3. Reference

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( )

Retruns:

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()

Returns the electrical interface for each bit of the port.

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.

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.
3. Reference

**digitalio→get_portSize()**

**digitalio→portSize()**

digitalio.get_portSize()

<table>
<thead>
<tr>
<th>Returns the number of bits implemented in the I/O port.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int get_portSize( )</td>
</tr>
</tbody>
</table>

**Returns:**

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

On failure, throws an exception or returns Y_PORTSIZE_INVALID.
3. Reference

**YDigitalIO**

digitalio.get_portState()

digitalio.portState() digitalio.get_portState()

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

```c
int get_portState()
```

**Returns:**

an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on.

On failure, throws an exception or returns **Y_PORTSTATE_INVALID**.
digitalio→get_userData() digitalio→userData() digitalio.get_userData()

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

Object get_userData()  

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

**Returns:**

the object stored previously by the caller.
digitalio→isOnline()\texttt{digitalio.isOnline()}\hspace{1cm} YDigitalIO

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

\textbf{boolean isOnline( )}

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

\textbf{Returns :}
\texttt{true} if the digital IO port can be reached, and \texttt{false} otherwise
Preloads the digital IO port cache with a specified validity duration.

```c
int load( long msValidity)
```

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

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

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

On failure, throws an exception or returns a negative error code.
digitalio.nextDigitalIO()  

Continues the enumeration of digital IO ports started using yFirstDigitalIO().

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.
3. Reference

digitalio\rightarrow pulse() digitalio.pulse() YDigitalIO

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()

digitalio.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

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

digitalio→set_bitDirection()
digitalio→setBitDirection()
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.
### digitalio->set_bitOpenDrain()
digitalio->setBitOpenDrain()
digitalio.set_bitOpenDrain()

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

```c
int set_bitOpenDrain( int bitno, int opendrain)
```

#### Parameters :
- `bitno` the bit number; lowest bit has index 0
- `opendrain` 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output.
  
  Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

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

On failure, throws an exception or returns a negative error code.
Changes the polarity of a single bit from the I/O port.

```c
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.
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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int <code>set_logicalName(String newval)</code></td>
<td>changes the logical name of the digital IO port.</td>
</tr>
</tbody>
</table>

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)
```

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.

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.
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.
Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

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

```
int set_portState(int newval)
```

This function has no effect on bits configured as input in `portDirection`.

**Parameters:**
- `newval`: an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

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

On failure, throws an exception or returns a negative error code.
digitalio→set_userData() YDigitalIO
digitalio→setUserData()
digitalio.set_userData() digitalio.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
digitalio.toggle_bitState()

digitalio.toggle_bitState()

Reverts a single bit of the I/O port.

int toggle_bitState( int bitno)

Parameters:
  *bitno* the bit number; lowest bit has index 0

Returns:
  *YAPI_SUCCESS* if the call succeeds.

On failure, throws an exception or returns a negative error code.
3.14. 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"
```

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

```
pas
uses yocto_display;
```

```
vb
yocto_display.vb
```

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

```
py
from yocto_display import *
```

### Global functions

**yFindDisplay(func)**

Retrieves a display for a given identifier.

**yFirstDisplay()**

Starts the enumeration of displays currently accessible.

### YDisplay methods

**display→copyLayerContent(srcLayerId, dstLayerId)**

Copies the whole content of a layer to another layer.

**display→describe()**

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

**display→fade(brightness, duration)**

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

**display→get_advertisedValue()**

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

**display→get_brightness()**

Returns the luminosity of the module informative leds (from 0 to 100).

**display→get_displayHeight()**

Returns the display height, in pixels.

**display→get_displayLayer(layerId)**

Returns a YDisplayLayer object that can be used to draw on the specified layer.

**display→get_displayType()**

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

**display→get_displayWidth()**

Returns the display width, in pixels.

**display→get_enabled()**

Returns true if the screen is powered, false otherwise.

**display→get_errorMessage()**

Returns the error message of the latest error with the display.
### 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 displayed is powered on.

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

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

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

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

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

**display**→newSequence()
- Starts to record all display commands into a sequence, for later replay.

**display**→nextDisplay()
- Continues the enumeration of displays started using yFirstDisplay().

**display**→pauseSequence(delay_ms)
- Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

**display**→playSequence(sequenceName)
- Replays a display sequence previously recorded using newSequence() and saveSequence().

**display**→registerValueCallback(callback)
3. Reference

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

```plaintext
display.resetAll()
```
Cleans the display screen and resets all display layers to their default state.

```plaintext
display.saveSequence(sequenceName)
```
Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```plaintext
display.set_brightness(newval)
```
Changes the brightness of the display.

```plaintext
display.set_enabled(newval)
```
Changes the power state of the display.

```plaintext
display.set_logicalName(newval)
```
Changes the logical name of the display.

```plaintext
display.set_orientation(newval)
```
Changes the display orientation.

```plaintext
display.set_startupSeq(newval)
```
Changes the name of the sequence to play when the display is powered on.

```plaintext
display.set_userData(data)
```
Stores a user context provided as argument in the userData attribute of the function.

```plaintext
display.stopSequence()
```
Stops immediately any ongoing sequence replay.

```plaintext
display.swapLayerContent(layerIdA, layerIdB)
```
Swaps the whole content of two layers.

```plaintext
display.upload(pathname, content)
```
Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

```plaintext
display.wait_async(callback, context)
```
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YDisplay.FindDisplay()

YDisplay.FindDisplay() retrieves a display for a given identifier.

Function YDisplay.FindDisplay( String func)

The identifier can be specified using several formats:

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

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

Parameters :
  - func a string that uniquely characterizes the display

Returns :
  - a YDisplay object allowing you to drive the display.
YDisplay.FirstDisplay()

YDisplay.FirstDisplay()

<table>
<thead>
<tr>
<th>YDisplay FirstDisplay( )</th>
</tr>
</thead>
</table>

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() YDisplay
defaultDisplay.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)
display.fade() 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.
display \rightarrow \text{get\_advertisedValue}() \\
\text{display} \rightarrow \text{advertisedValue}() \\
\text{display.get\_advertisedValue}() \\

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

\begin{Verbatim}
String \text{get\_advertisedValue}( )
\end{Verbatim}

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

On failure, throws an exception or returns \text{Y\_ADVERTISEDVALUE\_INVALID}. 
### YDisplay

- **display→get_brightness()**
- **display→brightness()**

`display.get_brightness()` returns the luminosity of the module informative leds (from 0 to 100).

```c
int get_brightness()
```

**Returns:**
- an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns **Y_BRIGHTNESS_INVALID**.
Returns the display height, in pixels.

int get_displayHeight() {
    \textbf{Returns :}
    \begin{itemize}
    \item an integer corresponding to the display height, in pixels
    \end{itemize}

    On failure, throws an exception or returns \texttt{Y_DISPLAYHEIGHT_INVALID}.
Returns a YDisplayLayer object that can be used to draw on the specified layer.

synchronized YDisplayLayer get_displayLayer( int layerId)

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

Parameters:
- **layerId** the identifier of the layer (a number in range 0..layerCount-1)

Returns:
- an YDisplayLayer object

On failure, throws an exception or returns null.
Returns the display type: monochrome, gray levels or full color.

Returns:

- a value among `Y_DISPLAYTYPE_MONO`, `Y_DISPLAYTYPE_GRAY` and `Y_DISPLAYTYPE_RGB` corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns `Y_DISPLAYTYPE_INVALID`. 
display → get_displayWidth()
display → displayWidth()
display.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.
3. Reference

`display.get_enabled()`  
`display->enabled()`  

Returns true if the screen is powered, false otherwise.

```c
int get_enabled()  

Returns:

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the screen is powered, false otherwise.

On failure, throws an exception or returns `Y_ENABLED_INVALID`.```
Returns the error message of the latest error with the display.

String `get_errorMessage()`

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

**Returns:**

a string corresponding to the latest error message that occurred while using the display object
display.get_errorType()

**YDisplay**

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

```c
int get_errorType()
```

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

**Returns:**

a number corresponding to the code of the latest error that occurred while using the display object
display \rightarrow \text{get\_friendlyName()}

\text{display} \rightarrow \text{friendlyName()}

\text{display.get\_friendlyName()}

Returns a global identifier of the display in the format \text{MODULE\_NAME.FUNCTION\_NAME}.

\text{String get\_friendlyName()}

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

Returns :

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

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

YDisplay

\text{display} \rightarrow \text{get\_friendlyName()}

\text{display} \rightarrow \text{friendlyName()}

\text{display.get\_friendlyName()}

Returns a global identifier of the display in the format \text{MODULE\_NAME.FUNCTION\_NAME}.

\text{String get\_friendlyName()}

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

Returns :

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

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

YDisplay
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>display.get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.</td>
</tr>
<tr>
<td><code>display.functionDescriptor()</code></td>
<td>This identifier can be used to test if two instances of <code>YFunction</code> reference the same physical function on the same physical device.</td>
</tr>
</tbody>
</table>

**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 display, without reference to the module.

**String get_functionId()**

For example `relay1`

**Returns :**

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

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
display.get_hardwareId() Returns the unique hardware identifier of the display in the form SERIAL.FUNCTIONID.

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

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

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
display\rightarrow\text{get\_layer\_Count}() \quad \text{YDisplay}
display\rightarrow\text{layer\_Count}() \quad \text{display.get\_layer\_Count}() \quad \text{YDisplay}

<table>
<thead>
<tr>
<th>Returns</th>
<th>an integer corresponding to the number of available layers to draw on</th>
</tr>
</thead>
</table>

On failure, throws an exception or returns \text{Y\_LAYER\_COUNT\_INVALID}. 

\text{int get\_layer\_Count()}
3. Reference

`display`→`get_layerHeight()`
`display`→`layerHeight()`
`display.get_layerHeight()`

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

```c
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`. 
### Reference

**YDisplay**

#### `display.get_layerWidth()`

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

#### `int get_layerWidth()`

**Returns:**

- an integer corresponding to the width of the layers to draw on, in pixels

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

display → get_logicalName()
display → logicalName()
display.get_logicalName()

YDisplay

Returns the logical name of the display.

String get_logicalName()

Returns:

a string corresponding to the logical name of the display.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
display→get_module() YDisplay
display→module() display.get_module()

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

YModule get_module()

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

Returns:

an instance of YModule
display\rightarrow\text{get\_orientation()} \hspace{1cm} \text{YDisplay}\hspace{1cm} \text{display\rightarrow\text{orientation}()} \hspace{1cm} \text{display.get\_orientation()} \\

<table>
<thead>
<tr>
<th>Returns the currently selected display orientation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{int _get_orientation()} \hspace{1cm} \text{Returns :}</td>
</tr>
<tr>
<td>\hspace{1cm} \text{a value among Y_ORIENTATION_LEFT, Y_ORIENTATION_UP, Y_ORIENTATION_RIGHT and Y_ORIENTATION_DOWN corresponding to the currently selected display orientation}</td>
</tr>
<tr>
<td>\hspace{1cm} \text{On failure, throws an exception or returns Y_ORIENTATION_INVALID.}</td>
</tr>
</tbody>
</table>
display→get_startupSeq()

display→startupSeq() display.get_startupSeq()

Returns the name of the sequence to play when the displayed is powered on.

String get_startupSeq()

Returns:
- a string corresponding to the name of the sequence to play when the displayed is powered on.

On failure, throws an exception or returns Y_STARTUPSEQ_INVALID.
YDisplay

display→get_userData()
display→userData(display.get_userData())

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

Object `get_userData()`

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

**Returns:**

the object stored previously by the caller.
YDisplay

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

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

```c
int load( long msValidity)
```

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

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

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

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

Starts to record all display commands into a sequence, for later replay.

```c
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.
Continues the enumeration of displays started using \texttt{yFirstDisplay()}.

\textbf{YDisplay} \texttt{nextDisplay( )}

\textbf{Returns :}

\begin{itemize}
  \item a pointer to a \texttt{YDisplay} object, corresponding to a display currently online, or a \texttt{null} pointer if there are no more displays to enumerate.
\end{itemize}
**display.pauseSequence()**  
`display.pauseSequence()`  
Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

```java
int pauseSequence( int delay_ms)
```

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

**Parameters :**
- `delay_ms` the duration to wait, in milliseconds

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

On failure, throws an exception or returns a negative error code.
Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

**Parameters:**
- `sequenceName` the name of the newly created sequence

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

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

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

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

display→resetAll()display.resetAll()

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

```c
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.
**display**→**saveSequence()**

`display.saveSequence()`

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

```java
int saveSequence(String sequenceName)
```

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

**Parameters :**

- `sequenceName` the name of the newly created sequence

**Returns :**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
display→set_brightness()
display→setBrightness()
display.set_brightness()

Changes the brightness of the display.

```c
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.
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.
display->set_logicalName()
display=setLogicalName()
display.set_logicalName()  

Changes the logical name of the display.

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

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

### Parameters :
- `newval` a string corresponding to the logical name of the display.

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

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

```java
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.

```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
display.stopSequence() \(\rightarrow\) display.stopSequence()

YDisplay

Stops immediately any ongoing sequence replay.

```
int stopSequence()
```

The display is left as is.

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

On failure, throws an exception or returns a negative error code.
display->swapLayerContent()
dl&play.swapLayerContent ()

Swaps the whole content of two layers.

int swapLayerContent(int layerIdA, int layerIdB)

The color and transparency of all the pixels from the two layers are swapped. This method only affects
the displayed content, but does not change any property of the layer objects. In particular, the visibility
of each layer stays unchanged. When used between one hidden layer and a visible layer, this method
makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support
(it is always completely opaque).

**Parameters:**
- `layerIdA` the first layer (a number in range 0..layerCount-1)
- `layerIdB` the second layer (a number in range 0..layerCount-1)

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

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

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

```java
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
yocto_display.vb
```

```cs
yocto_display.cs
```

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

```py
from yocto_display import *
```

### YDisplayLayer methods

- `displaylayer→clear()`
  Erases the whole content of the layer (makes it fully transparent).

- `displaylayer→clearConsole()`
  Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

- `displaylayer→consoleOut(text)`
  Outputs a message in the console area, and advances the console pointer accordingly.

- `displaylayer→drawBar(x1, y1, x2, y2)`
  Draws a filled rectangular bar at a specified position.

- `displaylayer→drawBitmap(x, y, w, bitmap, bgcol)`
  Draws a bitmap at the specified position.

- `displaylayer→drawCircle(x, y, r)`
  Draws an empty circle at a specified position.

- `displaylayer→drawDisc(x, y, r)`
  Draws a filled disc at a given position.

- `displaylayer→drawImage(x, y, imagename)`
  Draws a GIF image at the specified position.

- `displaylayer→drawPixel(x, y)`
  Draws a single pixel at the specified position.

- `displaylayer→drawRect(x1, y1, x2, y2)`
  Draws an empty rectangle at a specified position.

- `displaylayer→drawText(x, y, anchor, text)`
  Draws a text string at the specified position.

- `displaylayer→get_display()`
  Gets parent YDisplay.

- `displaylayer→get_displayHeight()`
  Returns the display height, in pixels.

- `displaylayer→get_displayWidth()`
  Returns the display width, in pixels.
<table>
<thead>
<tr>
<th><strong>displaylayer</strong></th>
<th><strong>Method</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>displaylayer</td>
<td>get_layerHeight()</td>
<td>Returns the height of the layers to draw on, in pixels.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>get_layerWidth()</td>
<td>Returns the width of the layers to draw on, in pixels.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>hide()</td>
<td>Hides the layer.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>lineTo(x, y)</td>
<td>Draws a line from current drawing pointer position to the specified position.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>moveTo(x, y)</td>
<td>Moves the drawing pointer of this layer to the specified position.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>reset()</td>
<td>Reverts the layer to its initial state (fully transparent, default settings).</td>
</tr>
<tr>
<td>displaylayer</td>
<td>selectColorPen(color)</td>
<td>Selects the pen color for all subsequent drawing functions, including text drawing.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>selectEraser()</td>
<td>Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>selectFont(fontname)</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>displaylayer</td>
<td>selectGrayPen(graylevel)</td>
<td>Selects the pen gray level for all subsequent drawing functions, including text drawing.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>setAntialiasingMode(mode)</td>
<td>Enables or disables anti-aliasing for drawing oblique lines and circles.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>setConsoleBackground(bgcol)</td>
<td>Sets up the background color used by the clearConsole function and by the console scrolling feature.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>setConsoleMargins(x1, y1, x2, y2)</td>
<td>Sets up display margins for the consoleOut function.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>setConsoleWordWrap(wordwrap)</td>
<td>Sets up the wrapping behaviour used by the consoleOut function.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>setLayerPosition(x, y, scrollTime)</td>
<td>Sets the position of the layer relative to the display upper left corner.</td>
</tr>
<tr>
<td>displaylayer</td>
<td>unhide()</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()  
YDisplayLayer
displaylayer.clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

int clearConsole()  

Returns:
YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
displaylayer\texttt{\rightarrow}consoleOut() \quad \text{YDisplayLayer}

\texttt{displaylayer.\text{consoleOut}()}\hfill

Outputs a message in the console area, and advances the console pointer accordingly.

\begin{verbatim}
int \textbf{consoleOut}( \texttt{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.

\begin{itemize}
  \item **Parameters** :
    \begin{itemize}
      \item \texttt{text} \ the \ message \ to \ display
    \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.
Draws a filled rectangular bar at a specified position.

```c
int drawBar(int x1, int y1, int x2, int y2)
```

Parameters:
- `x1` the distance from left of layer to the left border of the rectangle, in pixels
- `y1` the distance from top of layer to the top border of the rectangle, in pixels
- `x2` the distance from left of layer to the right border of the rectangle, in pixels
- `y2` the distance from top of layer to the bottom border of the rectangle, in pixels

Returns:
- `YAPI_SUCCESS` if the call succeeds.

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

**drawBitmap()**

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.
### drawCircle()

**YDisplayLayer**

displaylayer\rightarrow\text{drawCircle()}

displaylayer\text{}.\text{drawCircle}\text{()}

Draws an empty circle at a specified position.

```c
int \text{drawCircle}( \text{int } x, \text{int } y, \text{int } r)
```

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x) the distance from left of layer to the center of the circle, in pixels</td>
</tr>
<tr>
<td>(y) the distance from top of layer to the center of the circle, in pixels</td>
</tr>
<tr>
<td>(r) the radius of the circle, in pixels</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{YAPI_SUCCESS} if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
displaylayer->drawDisc()

displaylayer.drawDisc()

Draws a filled disc at a given position.

int drawDisc( int x, int y, int r)

Parameters:
- x the distance from left of layer to the center of the disc, in pixels
- y the distance from top of layer to the center of the disc, in pixels
- r the radius of the disc, in pixels

Returns:
- YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
displaylayer.drawImage() (YDisplayLayer)

displaylayer.drawImage()

Draws a GIF image at the specified position.

int drawImage( int x, int y, String imagename)

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

Parameters:

- x: the distance from left of layer to the left of the image, in pixels
- y: the distance from top of layer to the top of the image, in pixels
- imagename: the GIF file name

Returns:

- YAPI_SUCCESS if the call succeeds.

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

displaylayer->drawPixel()
displaylayer.drawPixel()

Draws a single pixel at the specified position.

int drawPixel( int x, int y)

**Parameters:**
- x: the distance from left of layer, in pixels
- y: the distance from top of layer, in pixels

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

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

```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.
Draws a text string at the specified position.

```c
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 `ALIGN` enumeration: `Y_ALIGN_TOP_LEFT`, `Y_ALIGN_CENTER_LEFT`, `Y_ALIGN_BASELINE_LEFT`, `Y_ALIGN_BOTTOM_LEFT`, `Y_ALIGN_TOP_CENTER`, `Y_ALIGN_CENTER`, `Y_ALIGN_BASELINE_CENTER`, `Y_ALIGN_BOTTOM_CENTER`, `Y_ALIGN_TOP_DECIMAL`, `Y_ALIGN_CENTER_DECIMAL`, `Y_ALIGN_BASELINE_DECIMAL`, `Y_ALIGN_BOTTOM_DECIMAL`, `Y_ALIGN_TOP_RIGHT`, `Y_ALIGN_CENTER_RIGHT`, `Y_ALIGN_BASELINE_RIGHT`, `Y_ALIGN_BOTTOM_RIGHT`.
- `text` the text string to draw

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

On failure, throws an exception or returns a negative error code.
displaylayer→get_display()

displaylayer→display()

displaylayer.get_display()

YDisplayLayer

Gets parent YDisplay.

YDisplay get_display()

Returns the parent YDisplay object of the current YDisplayLayer.

Returns:

an YDisplay object
### 3. Reference

**YDisplayLayer**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>displaylayer.get_displayHeight()</code></td>
<td>Returns the display height, in pixels.</td>
</tr>
<tr>
<td><code>int get_displayHeight()</code></td>
<td>Returns: an integer corresponding to the display height, in pixels.</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.</td>
</tr>
</tbody>
</table>
3. Reference

displaylayer→get_displayWidth()
displaylayer→displayWidth()
displaylayer.get_displayWidth()

YDisplayLayer

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.
<p>| displaylayer→get_layerHeight() | YDisplayLayer |</p>
<table>
<thead>
<tr>
<th>displaylayer→layerHeight()</th>
<th>displaylayer.get_layerHeight()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the height of the layers to draw on, in pixels.</td>
<td></td>
</tr>
</tbody>
</table>
| ```
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.
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.
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.
draws a line from current drawing pointer position to the specified position.

```
int lineTo(int x, int y)
```

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

**Parameters:**
- `x` the distance from left of layer to the end point of the line, in pixels
- `y` the distance from top of layer to the end point of the line, in pixels

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

On failure, throws an exception or returns a negative error code.
displaylayer->moveTo() displaylayer.moveTo() YDisplayLayer

Moves the drawing pointer of this layer to the specified position.

**int moveTo( int x, int y)**

**Parameters:**
- **x** the distance from left of layer, in pixels
- **y** the distance from top of layer, in pixels

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

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

Reverts the layer to its initial state (fully transparent, default settings).

```c
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.
3. Reference

**displaylayer->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.
3. Reference

**displaylayer→selectEraser()**  
*YDisplayLayer*

DisplayLayer.selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

```c
int selectEraser()
```

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

**Returns:**

- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
displaylayer->selectFont() YDisplayLayer
displaylayer.selectFont()

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```c
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 lightest 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.

```
int setAntialiasingMode( boolean mode)
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

**Parameters:**
- **mode** true to enable antialiasing, false to disable it.

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

On failure, throws an exception or returns a negative error code.
**displaylayer**→**setConsoleBackground()**  
YDisplayLayer  
displaylayer.setConsoleBackground()

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

```c
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.
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.
Sets up the wrapping behaviour used by the consoleOut function.

```java
int setConsoleWordWrap( boolean wordwrap)
```

**Parameters:**
- `wordwrap` true to wrap only between words, false to wrap on the last column anyway.

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

On failure, throws an exception or returns a negative error code.
**displaylayer→setLayerPosition()**

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.
3. Reference

`displaylayer.unhide()`

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. 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 *
```

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>yFindDualPower(func)</td>
</tr>
<tr>
<td>Retrieves a dual power control for a given identifier.</td>
</tr>
<tr>
<td>yFirstDualPower()</td>
</tr>
<tr>
<td>Starts the enumeration of dual power controls currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YDualPower methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>dualpower→describe()</td>
</tr>
<tr>
<td>Returns a short text that describes unambiguously the instance of the power control in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>dualpower→get_advertisedValue()</td>
</tr>
<tr>
<td>Returns the current value of the power control (no more than 6 characters).</td>
</tr>
<tr>
<td>dualpower→get_errorMessage()</td>
</tr>
<tr>
<td>Returns the error message of the latest error with the power control.</td>
</tr>
<tr>
<td>dualpower→get_errorType()</td>
</tr>
<tr>
<td>Returns the numerical error code of the latest error with the power control.</td>
</tr>
<tr>
<td>dualpower→get_extVoltage()</td>
</tr>
<tr>
<td>Returns the measured voltage on the external power source, in millivolts.</td>
</tr>
<tr>
<td>dualpower→get_friendlyName()</td>
</tr>
<tr>
<td>Returns a global identifier of the power control in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td>dualpower→get_functionDescriptor()</td>
</tr>
<tr>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td>dualpower→get_functionId()</td>
</tr>
<tr>
<td>Returns the hardware identifier of the power control, without reference to the module.</td>
</tr>
<tr>
<td>dualpower→get_hardwareId()</td>
</tr>
<tr>
<td>Returns the unique hardware identifier of the power control in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>dualpower→get_logicalName()</td>
</tr>
<tr>
<td>Returns the logical name of the power control.</td>
</tr>
<tr>
<td>dualpower→get_module()</td>
</tr>
</tbody>
</table>
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dualpower.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>dualpower.get_powerControl()</td>
<td>Returns the selected power source for module functions that require lots of current.</td>
</tr>
<tr>
<td>dualpower.get_powerState()</td>
<td>Returns the current power source for module functions that require lots of current.</td>
</tr>
<tr>
<td>dualpower.get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>dualpower.isOnline()</td>
<td>Checks if the power control is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>dualpower.isOnline_async(callback, context)</td>
<td>Checks if the power control is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>dualpower.load(msValidity)</td>
<td>Preloads the power control cache with a specified validity duration.</td>
</tr>
<tr>
<td>dualpower.load_async(msValidity, callback, context)</td>
<td>Preloads the power control cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>dualpower.nextDualPower()</td>
<td>Continues the enumeration of dual power controls started using yFirstDualPower().</td>
</tr>
<tr>
<td>dualpower.registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>dualpower.set_logicalName(newval)</td>
<td>Changes the logical name of the power control.</td>
</tr>
<tr>
<td>dualpower.set_powerControl(newval)</td>
<td>Changes the selected power source for module functions that require lots of current.</td>
</tr>
<tr>
<td>dualpower.set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>dualpower.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>
YDualPower.FindDualPower()

YDualPower.FindDualPower()

Retrieves a dual power control for a given identifier.

YDualPower.FindDualPower(String func)

Parameters:
- func a string that uniquely characterizes the power control

Returns:
a YDualPower object allowing you to drive the power control.
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()` Returns a short text that describes unambiguously the instance of the power control 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 power control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
YDualPower

dualpower→get_advertisedValue()
dualpower→advertisedValue()
dualpower.get_advertisedValue()

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

String get_advertisedValue() { }

Returns:

- a string corresponding to the current value of the power control (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
dualpower\rightarrow get\_errorMessage()
dualpower\rightarrow errorMessage()
dualpower.get\_errorMessage()

Returns the error message of the latest error with the power control.

String \textbf{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 power control object
dualpower→get_errorType()
dualpower→errorType()
dualpower.get_errorType()

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

int get_errorType()

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

Returns:
- a number corresponding to the code of the latest error that 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.

int get_extVoltage()  

Returns:

an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns Y_EXTVOLTAGE_INVALID.
dualpower→get_friendlyName()
dualpower→friendlyName()
dualpower.get_friendlyName()

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

String get_friendlyName( )

The returned string uses the logical names of the module and of the power control if they are defined, otherwise the serial number of the module and the hardware identifier of the power control (for example: MyCustomNamerelay1)

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.
YDualPower

dualpower → get_functionDescriptor()
dualpower → functionDescriptor()
dualpower.get_functionDescriptor()

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

String get_functionDescriptor()

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

Returns:

- an identifier of type YFUN_DESCR.

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

```java
YDualPower dualpower;
```
```java
dualpower.get_functionId();
```
```java
dualpower.functionId();
```
```java
dualpower.get_functionId();
```
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.
3. Reference

dualpower→get_logicalName()        YDualPower
dualpower→logicalName()             dualpower.get_logicalName()

<table>
<thead>
<tr>
<th>String get_logicalName( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns :</td>
</tr>
<tr>
<td>a string corresponding to the logical name of the power control.</td>
</tr>
<tr>
<td>On failure, throws an exception or returns Y_LOGICALNAME_INVALID.</td>
</tr>
</tbody>
</table>
**dualpower.get_module()**

**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
dualpower \rightarrow \text{get\_powerControl()}
dualpower \rightarrow \text{powerControl()}
dualpower.\text{get\_powerControl()}

Returns the selected power source for module functions that require lots of current.

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

**Returns:**

- a value among \texttt{Y\_POWERCONTROL\_AUTO}, \texttt{Y\_POWERCONTROL\_FROM\_USB}, \texttt{Y\_POWERCONTROL\_FROM\_EXT} and \texttt{Y\_POWERCONTROL\_OFF} corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns \texttt{Y\_POWERCONTROL\_INVALID}.
dualpower→get_powerState()
dualpower→powerState()
dualpower.get_powerState()

Returns the current power source for module functions that require lots of current.

int get_powerState()

**Returns:**

a value among Y_POWERSTATE_OFF, Y_POWERSTATE_FROM_USB and Y_POWERSTATE_FROM_EXT corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERSTATE_INVALID.
dualpower → get_userData()
dualpower.getUserData() dualpower.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.
dualpower→isOnline()dualpower.isOnline() Checks if the power control is currently reachable, without raising any error.

boolean isOnline( )

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

**Returns:**

true if the power control can be reached, and false otherwise
3. Reference

dualpower→load() YDualPower
dualpower.load()

Preloads the power control cache with a specified validity duration.

```c
int load( long msValidity)
```

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

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

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

On failure, throws an exception or returns a negative error code.
YDualPower nextDualPower()

Continues the enumeration of dual power controls started using YFirstDualPower().

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( UpdateCallback callback)
```

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

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
dualpower->set_logicalName()
dualpower->setLogicalName()
dualpower.set_logicalName()

Changes the logical name of the power control.

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

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

**Parameters :**
- `newval` a string corresponding to the logical name of the power control.

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

On failure, throws an exception or returns a negative error code.
Changes the selected power source for module functions that require lots of current.

**Parameters:**
- `newval` a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
dualpower.set_userData()  
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
3.17. 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:

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

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

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

```cpp
#include "yocto_files.h"
```

```m
#import "yocto_files.h"
```

```pas
uses yocto_files;
```

```vb
yocto_files.vb
```

```cs
yocto_files.cs
```

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

```py
from yocto_files import *
```

### Global functions

**yFindFiles()**

Retrieves a filesystem for a given identifier.

**yFirstFiles()**

Starts the enumeration of filesystems currently accessible.

### YFiles methods

**files→describe()**

Returns a short text that describes unambiguously the instance of the filesystem in the form

```
TYPE(NAME)=SERIAL.FUNCTIONID
```

**files→download(pathname)**

Downloads the requested file and returns a binary buffer with its content.

**files→download_async(pathname, callback, context)**

Downloads the requested file and returns a binary buffer with its content.

**files→format_fs()**

Reinitialize the filesystem to its clean, unfragmented, empty state.

**files→get_advertisedValue()**

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

**files→get_errorMessage()**

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

**files→get_errorType()**

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

**files→get_filesCount()**

Returns the number of files currently loaded in the filesystem.

**files→get_freeSpace()**

Returns the free space for uploading new files to the filesystem, in bytes.

**files→get_friendlyName()**

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

**files→get_functionDescriptor()**

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

**files→get_functionId()**
Returns the hardware identifier of the filesystem, without reference to the module.

```plaintext
files→get_hardwareId()
```

Returns the unique hardware identifier of the filesystem in the form SERIAL:FUNCTIONID.

```plaintext
files→get_list(pattern)
```

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```plaintext
files→get_logicalName()
```

Returns the logical name of the filesystem.

```plaintext
files→get_module()
```

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

```plaintext
files→get_module_async(callback, context)
```

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

```plaintext
files→get_userData()
```

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

```plaintext
files→isOnline()
```

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

```plaintext
files→isOnline_async(callback, context)
```

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

```plaintext
files→load(msValidity)
```

Preloads the filesystem cache with a specified validity duration.

```plaintext
files→load_async(msValidity, callback, context)
```

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

```plaintext
files→nextFiles()
```

Continues the enumeration of filesystems started using yFirstFiles().

```plaintext
files→registerValueCallback(callback)
```

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

```plaintext
files→remove(pathname)
```

Deletes a file, given by its full path name, from the filesystem.

```plaintext
files→set_logicalName(newval)
```

Changes the logical name of the filesystem.

```plaintext
files→set_userData(data)
```

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

```plaintext
files→upload(pathname, content)
```

Uploads a file to the filesystem, to the specified full path name.

```plaintext
files→wait_async(callback, context)
```

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

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.
3. Reference

files.describe() returns a short text that describes unambiguously the instance of the filesystem 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 filesystem (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
Reinitialize the filesystem to its clean, unfragmented, empty state.

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.
YFiles

files→get_advertisedValue()
files→advertisedValue()
files.get_advertisedValue()

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

String get_advertisedValue()

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
**files.get_errorMessage()**

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

```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 filesystem object
3. Reference

files → `get_errorType()`
files → `errorType()`

`files.get_errorType()`

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

```cpp
int get_errorType()
```

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

**Returns:**

a number corresponding to the code of the latest error that occurred while using the filesystem object
Files \to get\_filesCount()
\texttt{get\_filesCount}()
Files.get\_filesCount()

Returns the number of files currently loaded in the filesystem.

\texttt{int get\_filesCount( )}

\textbf{Returns}:

- an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns \texttt{Y\_FILESCOUNT\_INVALID}.
files\rightarrow get\_freeSpace() 
files\rightarrow freeSpace() files.get_freeSpace() 

Returns the free space for uploading new files to the filesystem, in bytes.

\textbf{int get\_freeSpace()} 

\textbf{Returns :}
\begin{itemize}
  \item an integer corresponding to the free space for uploading new files to the filesystem, in bytes
\end{itemize}

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

files→get_friendlyName()

`files.get_friendlyName()`

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

**String get_friendlyName()**

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for 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 YFUN_DESCR corresponding to the function.

String get_functionDescriptor()

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

**Returns :**

- an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTION_INVALID.
files\rightarrow get\_functionId() 
files\rightarrow functionId() files.get\_functionId()

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

String get\_functionId() 

For example relay1

**Returns:**

- a string that identifies the filesystem (ex: relay1)

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

```java
files.get_hardwareId()
```

Returns the unique hardware identifier of the filesystem 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 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`. 

3. Reference

**YFiles**

files→get_list()

files→list() \texttt{files.get_list()}

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.


d| ArrayList<YFileRecord> | get_list( String pattern) |
|------------------------|---------------------------|

**Parameters :**

- **pattern** an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

**Returns :**

- a list of YFileRecord objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.
files→get_logicalName() YFiles
files→logicalName()files.get_logicalName()

Returns the logical name of the filesystem.

String get_logicalName() (

**Returns**: a string corresponding to the logical name of the filesystem.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
files.get_module()

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
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.
\begin{center}
3. Reference
\end{center}

\textbf{files\rightarrow isOnline() files.isOnline()}

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

\begin{verbatim}
boolean isOnline()
\end{verbatim}

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.

\textbf{Returns :}
\begin{quote}
true if the filesystem can be reached, and false otherwise
\end{quote}
files\rightarrow load() \rightarrow files.load()

Preloads the filesystem cache with a specified validity duration.

```c
int load( long msValidity)
```

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

**Parameters :**

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

**Returns :**

- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
files\texttt{\rightarrow nextFiles()} files.nextFiles() YFiles

Continues the enumeration of filesystems started using \texttt{yFirstFiles()}.

\texttt{YFiles\ nextFiles()} \texttt{()}

\textbf{Returns :}

a pointer to a \texttt{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( UpdateCallback callback)
```

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

**Parameters:**

- **callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
Files → remove() \texttt{files.remove()} 

Deletes a file, given by its full path name, from the filesystem.

\texttt{int remove( String \texttt{pathname})}

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use \texttt{format_fs} to fully reinitialize the filesystem.

**Parameters:**
- \texttt{pathname} path and name of the file to remove.

**Returns:**
- \texttt{YAPI\textunderscore 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.

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

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

**Parameters :**
- **newval** a string corresponding to the logical name of the filesystem.

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

On failure, throws an exception or returns a negative error code.
files→set_userData()
files→setUserData()\texttt{files.set_userData()}\\

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

\texttt{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}:
\begin{itemize}
\item \texttt{data} any kind of object to be stored
\end{itemize}
files\rightarrow\text{upload()}\hspace{1em} \text{files.upload()}

Uploads a file to the filesystem, to the specified full path name.

```java
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.
3.18. 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;
require_once('yocto_genericsensor.php');
```

```cpp
#include "yocto_genericsensor.h"
```

```m
uses yocto_genericsensor;
```

```vb
yocto_genericsensor.vb
```

```cs
yocto_genericsensor.cs
```

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

```py
from yocto_genericsensor import *
```

### Global functions

- **yFindGenericSensor** *(func)*
  - Retrieves a generic sensor for a given identifier.

- **yFirstGenericSensor** *(func)*
  - Starts the enumeration of generic sensors currently accessible.

### YGenericSensor 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**(func)
  - Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **get_advertisedValue**(func)
  - Returns the current value of the generic sensor (no more than 6 characters).

- **get_currentRawValue**(func)
  - Returns the uncalibrated, unrounded raw value returned by the sensor.

- **get_currentValue**(func)
  - Returns the current measured value.

- **get_errorMessage**(func)
  - Returns the error message of the latest error with the generic sensor.

- **get_errorType**(func)
  - Returns the numerical error code of the latest error with the generic sensor.

- **get_friendlyName**(func)
  - Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.

- **get_functionDescriptor**(func)
  - Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **get_functionId**(func)
  - Returns the hardware identifier of the generic sensor, without reference to the module.

- **get_hardwareId**(func)
  - Returns the unique hardware identifier of the generic sensor in the form `SERIAL.FUNCTIONID`. 
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genericsensor→get_highestValue()</td>
<td>Returns the maximal value observed for the measure since the device was started.</td>
</tr>
<tr>
<td>genericsensor→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>genericsensor→get_logicalName()</td>
<td>Returns the logical name of the generic sensor.</td>
</tr>
<tr>
<td>genericsensor→get_lowestValue()</td>
<td>Returns the minimal value observed for the measure since the device was started.</td>
</tr>
<tr>
<td>genericsensor→get_module()</td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td>genericsensor→get_module_async(callback, context)</td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>genericsensor→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>genericsensor→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>genericsensor→get_resolution()</td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td>genericsensor→get_signalBias()</td>
<td>Returns the electric signal bias for zero shift adjustment.</td>
</tr>
<tr>
<td>genericsensor→get_signalRange()</td>
<td>Returns the electric signal range used by the sensor.</td>
</tr>
<tr>
<td>genericsensor→get_signalUnit()</td>
<td>Returns the measuring unit of the electrical signal used by the sensor.</td>
</tr>
<tr>
<td>genericsensor→get_signalValue()</td>
<td>Returns the measured value of the electrical signal used by the sensor.</td>
</tr>
<tr>
<td>genericsensor→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td>genericsensor→get_valueRange()</td>
<td>Returns the physical value range measured by the sensor.</td>
</tr>
<tr>
<td>genericsensor→isOnline()</td>
<td>Checks if the generic sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>genericsensor→isOnline_async(callback, context)</td>
<td>Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>genericsensor→load(msValidity)</td>
<td>Preloads the generic sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td>genericsensor→loadCalibrationPoints(rawValues, refValues)</td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td>genericsensor→load_async(msValidity, callback, context)</td>
<td>Preloads the generic sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>genericsensor→nextGenericSensor()</td>
<td>Continues the enumeration of generic sensors started using <code>yFirstGenericSensor()</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>registerTimedReportCallback</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>registerValueCallback</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>set_highestValue</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>set_logFrequency</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>set_logicalName</code></td>
<td>Changes the logical name of the generic sensor.</td>
</tr>
<tr>
<td><code>set_lowestValue</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>set_reportFrequency</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>set_resolution</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>set_signalBias</code></td>
<td>Changes the electric signal bias for zero shift adjustment.</td>
</tr>
<tr>
<td><code>set_signalRange</code></td>
<td>Changes the electric signal range used by the sensor.</td>
</tr>
<tr>
<td><code>set_unit</code></td>
<td>Changes the measuring unit for the measured value.</td>
</tr>
<tr>
<td><code>set_userData</code></td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><code>set_valueRange</code></td>
<td>Changes the physical value range measured by the sensor.</td>
</tr>
<tr>
<td><code>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>
<tr>
<td><code>zeroAdjust</code></td>
<td>Adjusts the signal bias so that the current signal value is need precisely as zero.</td>
</tr>
</tbody>
</table>
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.
3. Reference

genericsensor → calibrateFromPoints()

genericsensor.calibrateFromPoints()

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

```java
int calibrateFromPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

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

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

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

\text{genericsensor.describe()}

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

\textbf{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.

\textbf{Returns :}

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

```java
YGenericSensor

genericsensor→get_advertisedValue()

genericsensor→advertisedValue()

genericsensor.get_advertisedValue()
```

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

```java
String get_advertisedValue()
```

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

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
Returns the uncalibrated, unrounded raw value returned by the sensor.

```java
double get_currentRawValue()
```

Returns:
- A floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor.

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

genericsensor→get_currentValue() YGenericSensor
genericsensor→currentValue()
genericsensor.get_currentValue()  

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the current measured value</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.
Genericsensor\rightarrow \texttt{get\_errorMessage()}

Genericsensor\rightarrow \texttt{errorMessage()}

\texttt{genericsensor.get\_errorMessage()}

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

\textbf{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 generic sensor object
genericsensor->get_errorType()
genericsensor->errorType()
genericsensor.get_errorType()

YGenericSensor

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

int get_errorType()

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

Returns:
- a number corresponding to the code of the latest error that occurred while using the generic sensor object
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.
Returns a unique identifier of type \texttt{YFUN\_DESCR} corresponding to the function.

\begin{verbatim}
String 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.

**Returns:**

an identifier of type \texttt{YFUN\_DESCR}.

If the function has never been contacted, the returned value is \texttt{Y\_FUNCTIONDESCRIPTOR\_INVALID}.
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.
YGenericSensor

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.
genericsensor→get_highestValue()
YGenericSensor
genericsensor→highestValue()
genericsensor.get_highestValue()

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

double get_highestValue() 

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
Genericsensor → get_logFrequency() YGenericSensor
Genericsensor → logFrequency()
Genericsensor.get_logFrequency()

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

String get_logFrequency() {

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.
genericsensor→get_logicalName()
genericsensor→logicalName()
genericsensor.get_logicalName()

Returns the logical name of the generic sensor.

String get_logicalName( )

Returns :

a string corresponding to the logical name of the generic sensor.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
genericsensor→get_lowestValue()  
YGenericSensor  

genericsensor→lowestValue()  

genericsensor.get_lowestValue()  

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

double get_lowestValue( )

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
**get_module()**

`genericsensor.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`
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 measurement, 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 measurement, 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.
Return 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`. 
YGenericSensor

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 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.
YGenericSensor\n\ngenericsensor \rightarrow \text{get\_signalRange()}
\ngenericsensor \rightarrow \text{signalRange()}
\ngenericsensor.get\_signalRange()

Returns the electric signal range used by the sensor.

\textbf{String get\_signalRange( )}

\textbf{Returns :}
\begin{itemize}
  \item a string corresponding to the electric signal range used by the sensor
\end{itemize}

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

genericsensor → get_signalUnit()
genericsensor → signalUnit()
genericsensor.get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

String get_signalUnit()  

Returns:
   a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALUNIT_INVALID.
3. Reference

genericsensor\rightarrow get\_signalValue()
genericsensor\rightarrow signalValue()
genericsensor.get\_signalValue()

Returns the measured value of the electrical signal used by the sensor.

double get\_signalValue()

Returns:

- a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns Y\_SIGNALVALUE\_INVALID.
get_unit() Returns the measuring unit for the measure.

String get_unit()

Returns:
- a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns Y_UNIT_INVALID.
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**Returns:**
- the object stored previously by the caller.
genericsensor\rightarrow \textit{get\_valueRange()}

\texttt{genericsensor valueRange()}

\texttt{genericsensor.get\_valueRange()}

Returns the physical value range measured by the sensor.

\begin{tabular}{|l|}
\hline
\texttt{String \textit{get\_valueRange()}} \\
\hline
\end{tabular}

\textbf{Returns :}

a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns \texttt{Y\_VALUERANGE\_INVALID}. 
### YGenericSensor

Check if the generic sensor is currently reachable, without raising any error.

#### boolean isOnline()

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

**Returns:**

- **true** if the generic sensor can be reached, and **false** otherwise
Preloads the generic sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

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

**Parameters :**

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

**Returns :**

- `YAPI_SUCCESS` when the call succeeds.

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

genericsensor.loadCalibrationPoints()

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

```java
int loadCalibrationPoints( ArrayList<Double> rawValues,
                          ArrayList<Double> refValues)
```

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

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

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

\text{nextGenericSensor()}

Continues the enumeration of generic sensors started using \text{yFirstGenericSensor()}.

Returns:
a pointer to a \text{YGenericSensor} object, corresponding to a generic sensor currently online, or a null pointer if there are no more generic sensors to enumerate.
Genericsensor → registerTimedReportCallback()  YGenericSensor
genericsensor.registerTimedReportCallback( )

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

```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( UpdateCallback callback)
```

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

**Parameters**:
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
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.
### genericsensor → set_logicalName()

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.
genericsensor->set_lowestValue()
genericsensor->setLowestValue()
genericsensor.set_lowestValue()

Changes the recorded minimal value observed.

**int set_lowestValue( double newval)**

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

**Returns:**
- **YAPI_SUCCESS** if the call succeeds.
- On failure, throws an exception or returns a negative error code.
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.
Changes the electric signal bias for zero shift adjustment.

If your electric signal reads positive when it should be zero, set a positive signal bias of the same value to fix the zero shift.

**Parameters:**
- `newval` a floating point number corresponding to the electric signal bias for zero shift adjustment

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

On failure, throws an exception or returns a negative error code.
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.
genericsensor.set_unit()  
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.
YGenericSensor

**genericsensor.set_userData()**

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.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
</tbody>
</table>
Genericsensor→set_valueRange()
genericsensor→setValueRange()
genericsensor.set_valueRange()

Changes the physical value range measured by the sensor.

```c
int set_valueRange( 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.
genericsensor→zeroAdjust()  

genericsensor.zeroAdjust()  

Adjusts the signal bias so that the current signal value is need precisely as zero.

int zeroAdjust()  

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;
```

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

```cpp
#include "yocto_gyro.h"
```

```m
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 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**()
3. Reference

Returns the unique hardware identifier of the gyroscope in the form SERIAL_FUNCTIONID.

**gyro→get_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_highestValue()**

Returns the maximal value observed for the angular velocity since the device was started.

**gyro→get_logFrequency()**

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

**gyro→get_logicalName()**

Returns the logical name of the gyroscope.

**gyro→get_lowestValue()**

Returns the minimal value observed for the angular velocity since the device was started.

**gyro→get_module()**

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

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

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

**gyro→get_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionW()**

Returns the $w$ component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionX()**

Returns the $x$ component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionY()**

Returns the $y$ component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_quaternionZ()**

Returns the $z$ component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

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

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

**gyro→get_reportFrequency()**

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

**gyro→get_resolution()**

Returns the resolution of the measured values.

**gyro→get_roll()**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

**gyro→get_unit()**

Returns the measuring unit for the angular velocity.

**gyro→get_userData()**

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

**gyro→get_xValue()**
3. Reference

- 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.FindGyro()

Retrieves a gyroscope for a given identifier.

YGyro.FindGyro(String func)

The identifier can be specified using several formats:

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

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

Parameters:

- func a string that uniquely characterizes the gyroscope

Returns:

- a YGyro object allowing you to drive the gyroscope.
YGyro.FirstGyro()

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()**

---

**GYyro**

**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(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

<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.
### 3. Reference

**gyro.describe()**

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).

String get_advertisedValue()  

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

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

**Returns:**
- a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`. 
gyro \rightarrow \text{get}\_\text{currentValue}()  \\
\text{gyro} \rightarrow \text{currentValue}()  \\
gyro.get\_\text{currentValue}()  \\

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

double \text{get}\_\text{currentValue}()  \\

Returns:

- a floating point number corresponding to the current value of the angular velocity, in degrees per second, as a floating point number

On failure, throws an exception or returns \text{Y\_CURRENTVALUE\_INVALID}.
Returns the error message of the latest error with the gyroscope.

**String get_errorMessage()**

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

**Returns:**

a string corresponding to the latest error message that occurred while using the gyroscope object
Returns the numerical error code of the latest error with the gyroscope.

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
3. Reference

**gyro→get_friendlyName()**

**gyro→friendlyName()**

**gyro.get_friendlyName()**

YGyro

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

**String get_friendlyName()**

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

**Returns:**

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

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

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

```
String get_functionDescriptor( )
```

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

**Returns:**  
an identifier of type **YFUN_DESCR**.

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

`gyro->get_functionId()`

`gyro->functionId()`

`gyro.get_functionId()`

Y Gyro

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

String `get_functionId()`

For example `relay1`

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

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

**YGyro**

`gyro.get_hardwareId()`  
`gyro.hardwareId()`  
`gyro.get_hardwareId()`

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

**String get_hardwareId( )**

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

**Returns :**

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

**gyro→get_heading()**

**gyro→heading()**

**gyro.get_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```java
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

**YGyro**

`gyro.get_highestValue()`

Returns the maximal value observed for the angular velocity since the device was started.

```java
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`.  

---
3. Reference

**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`. 
Returns the logical name of the gyroscope.

\[
\text{String } \text{get\_logical\_Name}( )
\]

\textbf{Returns :}

a string corresponding to the logical name of the gyroscope.

On failure, throws an exception or returns \text{Y\_LOGICAL\_NAME\_INVALID}.
Returns the minimal value observed for the angular velocity since the device was started.

**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.
Getting a module:

```
YModule get_module()
```

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

**Returns:**

an instance of YModule
YGyro

\texttt{gyro\rightarrow get\_pitch()}
\texttt{gyro\rightarrow pitch() gyro.get\_pitch()}

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

\textbf{double get\_pitch()}\texttt{( )}

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 \texttt{YRefFrame}.

\textbf{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.

**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.

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()`

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.

```java
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.
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

**get_reportFrequency()**

**reportFrequency()**

**gyro.get_reportFrequency()**

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

**String get_reportFrequency()**

**Returns:**

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

On failure, throws an exception or returns **Y_REPORTFREQUENCY_INVALID**.
gyro->get_resolution()
gyro->resolution() gyro.get_resolution()

Returns the resolution of the measured values.

double get_resolution()

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

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.
3. Reference

`gyro.get_roll()`

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
double get_roll()
```

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

**Returns:**

A floating-point number corresponding to roll angle in degrees, between -180 and +180.
Returns the measuring unit for the angular velocity.

String get_unit()

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

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

**YGyro**

- **gyro→get_userData()**
- **gyro→userData()**

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

<table>
<thead>
<tr>
<th>Object get_userData()</th>
</tr>
</thead>
<tbody>
<tr>
<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>

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>the object stored previously by the caller.</td>
</tr>
</tbody>
</table>
Returns the angular velocity around the X axis of the device, as a floating point number.

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`.  

<table>
<thead>
<tr>
<th>gyro</th>
<th><code>get_xValue()</code></th>
<th>YGyro</th>
</tr>
</thead>
<tbody>
<tr>
<td>gyro</td>
<td><code>xValue()</code></td>
<td>gyro.get_xValue()</td>
</tr>
</tbody>
</table>

```c
double get_xValue() {
    // Implementation details...
    return angular_velocity_x;
}
```
### 3. Reference

**YGyro**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gyro→get_yValue()</td>
<td>Returns the angular velocity around the Y axis of the device, as a floating point number.</td>
</tr>
<tr>
<td>gyro→yValue() gyro . get_yValue()</td>
<td></td>
</tr>
</tbody>
</table>

**double get_yValue()**

**Returns:**

- a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns Y_VALUE_INVALID.
3. Reference

`gyro->get_zValue()`
`gyro->zValue()`
`gyro.get_zValue()`

Returns the angular velocity around the Z axis of the device, as a floating point number.

```cpp
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`. 
}`
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
**gyro.load() (gyro.load())**

Preloads the gyroscope cache with a specified validity duration.

```c
int load( long msValidity)
```

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

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

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

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

gyro\rightarrow {\texttt{loadCalibrationPoints()}}

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

\begin{verbatim}
int loadCalibrationPoints( ArrayList<Double> rawValues,
                          ArrayList<Double> refValues)
\end{verbatim}

**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 gyroscopes started using `yFirstGyro()`.

**YGyro**  

**YGyro**  

**Returns:**  
a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.
3. Reference

**gyro**\rightarrow**registerAnglesCallback()**

`gyro.registerAnglesCallback()` 

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

**YGyro**

**gyro**\rightarrow\text{registerQuaternionCallback()}\text{\hspace{1cm} YGyro}

gyro.registerQuaternionCallback()

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

\begin{verbatim}
int registerQuaternionCallback( YQuatCallback callback)
\end{verbatim}

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.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>callback</strong> 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).</td>
</tr>
</tbody>
</table>
3. Reference

`gyro.registerTimedReportCallback()`  

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

```
int registerTimedReportCallback(TimedReportCallback callback)
```

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

**Parameters:**

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

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

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

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.

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 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.
 gyrọ→set_lowestValue()
gyro→setLowestValue() gyro.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.
3. Reference

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.
gyro->set_resolution()
gyro->setResolution()gyro.set_resolution()

Changes the resolution of the measured physical values.

int set_resolution(double newval)

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

**Parameters:**

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

**Returns:**

*YAPI_SUCCESS* if the call succeeds.

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

**YGyro**

gyro → set_userData()

gyro → setUserData()
gyro.set_userData()

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
3.20. 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

- **yFindHubPort(func)**
  - Retrieves a Yocto-hub port for a given identifier.

- **yFirstHubPort()**
  - Starts the enumeration of Yocto-hub ports currently accessible.

### YHubPort methods

- **hubport → describe()**
  - Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **hubport → get_advertisedValue()**
  - Returns the current value of the Yocto-hub port (no more than 6 characters).

- **hubport → get_baudRate()**
  - Returns the current baud rate used by this Yocto-hub port, in kbps.

- **hubport → get_enabled()**
  - Returns true if the Yocto-hub port is powered, false otherwise.

- **hubport → get_errorMessage()**
  - Returns the error message of the latest error with the Yocto-hub port.

- **hubport → get_errorType()**
  - Returns the numerical error code of the latest error with the Yocto-hub port.

- **hubport → get_friendlyName()**
  - Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

- **hubport → get_functionDescriptor()**
  - Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **hubport → get_functionId()**
  - Returns the hardware identifier of the Yocto-hub port, without reference to the module.

- **hubport → get_hardwareId()**
  - Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

- **hubport → get_logicalName()**
  - Returns the logical name of the Yocto-hub port.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hubport→get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>hubport→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>hubport→get_portState()</code></td>
<td>Returns the current state of the Yocto-hub port.</td>
</tr>
<tr>
<td><code>hubport→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>hubport→isOnline()</code></td>
<td>Checks if the Yocto-hub port is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>hubport→isOnline_async(callback, context)</code></td>
<td>Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>hubport→load(msValidity)</code></td>
<td>Preloads the Yocto-hub port cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>hubport→load_async(msValidity, callback, context)</code></td>
<td>Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>hubport→nextHubPort()</code></td>
<td>Continues the enumeration of Yocto-hub ports started using <code>yFirstHubPort()</code>.</td>
</tr>
<tr>
<td><code>hubport→registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>hubport→set_enabled(newval)</code></td>
<td>Changes the activation of the Yocto-hub port.</td>
</tr>
<tr>
<td><code>hubport→set_logicalName(newval)</code></td>
<td>Changes the logical name of the Yocto-hub port.</td>
</tr>
<tr>
<td><code>hubport→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>hubport→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>
YHubPort.FindHubPort()

Retrieves a Yocto-hub port for a given identifier.

YHubPort.FindHubPort(String func)

The identifier can be specified using several formats:

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

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

Parameters:

- **func**: a string that uniquely characterizes the Yocto-hub port

Returns:

- a YHubPort object allowing you to drive the Yocto-hub port.
YHubPort.FirstHubPort()

```
YHubPort.FirstHubPort()
```

Starts the enumeration of Yocto-hub ports currently accessible.

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()  

YHubPort  

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

String describe( )  

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

<table>
<thead>
<tr>
<th>Returns true if the Yocto-hub port is powered, false otherwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int get_enabled( )</td>
</tr>
</tbody>
</table>

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→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()  

YHubPort

Returns the numerical error code of the latest error with the Yocto-hub port.

int get\_errorType()  

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

**Returns:**

a number corresponding to the code of the latest error that occurred while using the Yocto-hub port object
hubport→get_friendlyName()
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.

String `get_functionDescriptor()`

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

Returns:
- an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
hubport\rightarrow{\text{get\_functionId}}() \quad \text{YHubPort}

hubport\rightarrow{\text{functionId}}() \quad \text{hubport\_get\_functionId}()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

\textbf{String} \textbf{get\_functionId()}\quad \textbf{()}

For example \textit{relay1}

\textbf{Returns :}

a string that identifies the Yocto-hub port (ex: \textit{relay1})

On failure, throws an exception or returns \textbf{Y\_FUNCTIONID\_INVALID}. 
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\rightarrow get\_logicalName()\nhubport\rightarrow logicalName()\nhubport.get\_logicalName()\n
Returns the logical name of the Yocto-hub port.

String get\_logicalName( )

Returns :
  a string corresponding to the logical name of the Yocto-hub port.

On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.
hubport→get_module()  
hubport→module() hubport.get_module()  

YHubPort

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

YModule get_module()  

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

Returns:

an instance of YModule
hubport→get_portState()  YHubPort
hubport→portState() hubport.get_portState()

Returns the current state of the Yocto-hub port.

int get_portState( )

Returns: a value among Y_PORTSTATE_OFF, Y_PORTSTATE_OVRLD, Y_PORTSTATE_ON, Y_PORTSTATE_RUN and Y_PORTSTATE_PROG corresponding to the current state of the Yocto-hub port.

On failure, throws an exception or returns Y_PORTSTATE_INVALID.
YHubPort

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.
hubport → isOnline() hubport.isOnline() YHubPort

Checks if the Yocto-hub port is currently reachable, without raising any error.

boolean isOnline()

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

Returns:
true if the Yocto-hub port can be reached, and false otherwise
hubport->load()hubport.load() YHubPort

Preloads the Yocto-hub port cache with a specified validity duration.

int load( long msValidity)

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

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

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

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

<table>
<thead>
<tr>
<th><code>hubport.nextHubPort()</code></th>
</tr>
</thead>
</table>
| Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

**Returns:**

A pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a null pointer if there are no more Yocto-hub ports to enumerate.
hubport.registerValueCallback() YHubPort

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
### Changes the activation of the Yocto-hub port.

#### int set_enabled(int newval)

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code> either <code>Y_ENABLED_FALSE</code> or <code>Y_ENABLED_TRUE</code>, according to the activation of the Yocto-hub port</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 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.
hubport→set_userData()
hubport→setUserData()
hubport.set_userData()

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
3.21. 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
#include "yocto_humidity.h"
```
```pas
uses yocto_humidity;
```
```vb
yocto_humidity.vb
```
```cs
yocto_humidity.cs
```
```java
import com.yoctopuce.YoctoAPI.YHumidity;
```
```py
from yocto_humidity import *
```

### Global functions

- `yFindHumidity(func)`
  - Retrieves a humidity sensor for a given identifier.

- `yFirstHumidity()`
  - Starts the enumeration of humidity sensors currently accessible.

### YHumidity methods

- `humidity→calibrateFromPoints(rawValues, refValues)`
  - Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

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

- `humidity→get_advertisedValue()`
  - Returns the current value of the humidity sensor (no more than 6 characters).

- `humidity→get_currentRawValue()`
  - Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.

- `humidity→get_currentValue()`
  - Returns the current value of the humidity, in %RH, as a floating point number.

- `humidity→get_errorMessage()`
  - Returns the error message of the latest error with the humidity sensor.

- `humidity→get_errorType()`
  - Returns the numerical error code of the latest error with the humidity sensor.

- `humidity→get_friendlyName()`
  - Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

- `humidity→get_functionDescriptor()`
  - Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- `humidity→get_functionId()`
  - Returns the hardware identifier of the humidity sensor, without reference to the module.

- `humidity→get_hardwareId()`
  - Returns the unique hardware identifier of the humidity sensor in the form `SERIAL.FUNCTIONID`.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>humidity.get_highestValue()</td>
<td>Returns the maximal value observed for the humidity since the device was started.</td>
</tr>
<tr>
<td>humidity.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>humidity.get_logicalName()</td>
<td>Returns the logical name of the humidity sensor.</td>
</tr>
<tr>
<td>humidity.get_lowestValue()</td>
<td>Returns the minimal value observed for the humidity since the device was started.</td>
</tr>
<tr>
<td>humidity.get_module()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>humidity.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>humidity.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>humidity.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>humidity.get_resolution()</td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td>humidity.get_unit()</td>
<td>Returns the measuring unit for the humidity.</td>
</tr>
<tr>
<td>humidity.get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>humidity.isOnline()</td>
<td>Checks if the humidity sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>humidity.isOnline_async(callback, context)</td>
<td>Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>humidity.load(msValidity)</td>
<td>Preloads the humidity sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td>humidity.loadCalibrationPoints(rawValues, refValues)</td>
<td>Retrieves error correction data points previously entered using the method calibrateFromPoints.</td>
</tr>
<tr>
<td>humidity.load_async(msValidity, callback, context)</td>
<td>Preloads the humidity sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>humidity.nextHumidity()</td>
<td>Continues the enumeration of humidity sensors started using yFirstHumidity().</td>
</tr>
<tr>
<td>humidity.registerTimedReportCallback(callback)</td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td>humidity.registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>humidity.set_highestValue(newval)</td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td>humidity.set_logFrequency(newval)</td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td>humidity.set_logicalName(newval)</td>
<td>Changes the logical name of the humidity sensor.</td>
</tr>
</tbody>
</table>
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>humidity(\rightarrow) set_lowestValue(newval)</td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td>humidity(\rightarrow) set_reportFrequency(newval)</td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td>humidity(\rightarrow) set_resolution(newval)</td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td>humidity(\rightarrow) set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>humidity(\rightarrow) 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>
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()  
YHumidity.FirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

YHumidity FirstHumidity()

Use the method YHumidity.nextHumidity() to iterate on next humidity sensors.

**Returns:**

a pointer to a YHumidity object, corresponding to the first humidity sensor currently online, or a null pointer if there are none.
humidity->calibrateFromPoints()

humidity.calibrateFromPoints()

YHumidity

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

```
int calibrateFromPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

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

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

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

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

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

YHumidity

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

String describe()

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

Returns:

a string that describes the humidity sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
humidity→get advertisedValue()
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.
Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.

\[
\text{double } \text{get\_currentRawValue}( )
\]

**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 \text{Y\_CURRENTRAWVALUE\_INVALID}. 

```python
YHumidity
humidity \rightarrow \text{get\_currentRawValue()}
humidity \rightarrow \text{currentRawValue()}
humidity.get\_currentRawValue()
```
humidity.get_currentValue()

Returns the current value of the humidity, in %RH, as a floating point number.

```java
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`. 
### 3. Reference

**YHumidity**

**humidity.get_errorMessage()**

**humidity.errorMessage()**

**humidity.getMessage()**

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

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

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

**Returns**:

- a string corresponding to the latest error message that occurred while using the humidity sensor object
`humidity.get_errorType()`

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

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
3. Reference

YHumidity

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.
humidity→get_functionDescriptor()  
humidity→functionDescriptor()  
humidity.get_functionDescriptor()  

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

String get_functionDescriptor( )

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

Returns:

an identifier of type YFUN_DESCR.

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

**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`. 
3. Reference

YHumidity

humidity→get_hardwareId()
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.
Returns the maximal value observed for the humidity since the device was started.

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

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

humidity.get_logFrequency()

humidity.get_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.
humidity->get_lowestValue()
humidity->lowestValue()
humidity.get_lowestValue()

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

`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`. 

YHumidity
YHumidity

humidity → get_module()
humidity → module() humidity.get_module()

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

YModule get_module()

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

**Returns:**

- an instance of YModule
YHumidity

humidity→get_recordedData()
humidity→recordedData()
humidity.get_recordedData()

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

YDataSet get_recordedData( long startTime, long endTime)

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

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

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startTime</strong> 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> 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>

<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>
3. Reference

YHumidity

humidity.get_reportFrequency()  

humidity.reportFrequency()  

humidity.get_reportFrequency()

| YHumidity
| humidity
| get_reportFrequency() |
| 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()
humidity.resolution()
humidity.get_resolution()

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`.
YHumidity

humidity → get_unit()
humidity → unit() humidity.get_unit()

Returns the measuring unit for the humidity.

String get_unit()  

Returns:
- a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns Y_UNIT_INVALID.
Returns the value of the userData attribute, as previously stored using method \texttt{set\_userData}.

\textbf{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() humidity.isOnline()  YHumidity

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

boolean isOnline()

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

**Returns:**
true if the humidity sensor can be reached, and false otherwise
humidity->load() humidity.load()

Preloads the humidity sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

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

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

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

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

humidity->loadCalibrationPoints()

humidity.loadCalibrationPoints()

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

```java
int loadCalibrationPoints(ArrayList<Double> rawValues,
                          ArrayList<Double> refValues)
```

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

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

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

`humidity->nextHumidity()`  
`humidity.nextHumidity()`

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

**YHumidity nextHumidity( )**

**Returns**:

A pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a null pointer if there are no more humidity sensors to enumerate.
humidity.registerTimedReportCallback()

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.
YHumidity humidity.registerValueCallback()

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

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

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 humidity sensor.

### Parameters:
- `newval`: a string corresponding to the logical name of the humidity sensor.

### Returns:
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
humidity\rightarrow\text{set\_lowestValue}()

Changes the recorded minimal value observed.

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

\textbf{Parameters :}
\begin{itemize}
\item \textbf{newval} a floating point number corresponding to the recorded minimal value observed
\end{itemize}

\textbf{Returns :}
\begin{itemize}
\item \text{YAPI\_SUCCESS} if the call succeeds.
\item On failure, throws an exception or returns a negative error code.
\end{itemize}
humidity.set_reportFrequency()

Changes the timed value notification frequency for this function.

```cpp
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

humidity.set_resolution()  

YHumidity

humidity.setResolution()  

humidity.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.
3. Reference

humidity\rightarrow\texttt{set\_userData()}

humidity\rightarrow\texttt{setUserData()}

\texttt{humidity.set\_userData()}

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

\texttt{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 :}

\texttt{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:

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

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

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

```cpp
#include "yocto_led.h"
```

```m
#include "yocto_led.h"
```

```pas
uses yocto_led;
```

```vb
yocto_led.vb
```

```cs
yocto_led.cs
```

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

```py
from yocto_led import *
```

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindLed(func)</code></td>
</tr>
<tr>
<td><code>yFirstLed()</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YLed methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>led→describe()</code></td>
</tr>
<tr>
<td><code>led→get_advertisedValue()</code></td>
</tr>
<tr>
<td><code>led→get_blinking()</code></td>
</tr>
<tr>
<td><code>led→get_errorMessage()</code></td>
</tr>
<tr>
<td><code>led→get_errorType()</code></td>
</tr>
<tr>
<td><code>led→get_friendlyName()</code></td>
</tr>
<tr>
<td><code>led→get_functionDescriptor()</code></td>
</tr>
<tr>
<td><code>led→get_functionId()</code></td>
</tr>
<tr>
<td><code>led→get_hardwareId()</code></td>
</tr>
<tr>
<td><code>led→get_logicalName()</code></td>
</tr>
<tr>
<td><code>led→get_luminosity()</code></td>
</tr>
<tr>
<td><code>led→get_module()</code></td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td><code>get_module_async(callback, context)</code></td>
</tr>
<tr>
<td><code>get_power()</code></td>
</tr>
<tr>
<td><code>get_userData()</code></td>
</tr>
<tr>
<td><code>isOnline()</code></td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
</tr>
<tr>
<td><code>nextLed()</code></td>
</tr>
<tr>
<td><code>registerValueCallback(callback)</code></td>
</tr>
<tr>
<td><code>set_blinking(newval)</code></td>
</tr>
<tr>
<td><code>set_logicalName(newval)</code></td>
</tr>
<tr>
<td><code>set_luminosity(newval)</code></td>
</tr>
<tr>
<td><code>set_power(newval)</code></td>
</tr>
<tr>
<td><code>set_userData(data)</code></td>
</tr>
<tr>
<td><code>wait_async(callback, context)</code></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.
3. Reference

**YLed.FirstLed()**

Starts the enumeration of leds currently accessible.

Use the method **YLed.nextLed()** to iterate on next leds.

**Returns:**

- a pointer to a **YLed** object, corresponding to the first led currently online, or a null pointer if there are none.
led.describe() → led.describe()

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

String describe()

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

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

`led→get_advertisedValue()`

`led→advertisedValue()`

`led.get_advertisedValue()`

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

String `get_advertisedValue()`

Returns:

- a string corresponding to the current value of the led (no more than 6 characters).

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

**YLed**

- `get_blinking()`

  Returns the current led signaling mode.

- `int get_blinking()`

<table>
<thead>
<tr>
<th>Returns:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a value among <code>Y_BLINKING_STILL</code>, <code>Y_BLINKING_RELAX</code>, <code>Y_BLINKING_AWARE</code>, <code>Y_BLINKING_RUN</code>, <code>Y_BLINKING_CALL</code> and <code>Y_BLINKING_PANIC</code> corresponding to the current led signaling mode</td>
<td></td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_BLINKING_INVALID`. 
Returns the error message of the latest error with the led.

**String getErrorMessage()**

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

**Returns:**

- a string corresponding to the latest error message that occurred while using the led object
Returns the numerical error code of the latest error with the led.

int get_errorType()

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

Returns:

a number corresponding to the code of the latest error that occurred while using the led object
#### 3. Reference

**YLed**

`led.get_friendlyName()`

`led.friendlyName()`

`led.get_friendlyName()`

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

```java
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: `MyCustomName.relay1`)

**Returns:**

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

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

`String get_functionDescriptor()`

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

**Returns:**

an identifier of type `YFUN_DESCR`.

*If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`. 
Returns the hardware identifier of the led, without reference to the module.

String `get_functionId()`

For example `relay1`

**Returns:**
- a string that identifies the led (ex: `relay1`)
- On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.
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.
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**.
Returns the current led intensity (in per cent).

int get_luminosity()

Returns:
- an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.
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

**YLed**

`led->get_power()`

`led->power()`

`led.get_power()`

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`. 
3. Reference

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

<table>
<thead>
<tr>
<th>Object <code>get_userData</code>()</th>
</tr>
</thead>
</table>

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.
YLed

led.isOnline() Checks if the led is currently reachable, without raising any error.

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

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

```c
int load(long msValidity)
```

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

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

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

On failure, throws an exception or returns a negative error code.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nextLed()</code></td>
<td>Continues the enumeration of leds started using <code>yFirstLed()</code>.</td>
</tr>
<tr>
<td>Returns</td>
<td>a pointer to a <code>YLed</code> object, corresponding to a led currently online, or a null pointer if there are no more leds to enumerate.</td>
</tr>
</tbody>
</table>
3. Reference

**led.registerValueCallback()**

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

```c
int registerValueCallback( UpdateCallback callback)
```

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

**Parameters:**

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

```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 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).

**Parameters:**
- **newval** an integer corresponding to the current led intensity (in per cent)

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

On failure, throws an exception or returns a negative error code.
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.
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
3.23. 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
<script type='text/javascript' src='yocto_lightsensor.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YLightSensor = yoctolib.YLightSensor;
```
```php
require_once('yocto_lightsensor.php');
```
```cpp
#include "yocto_lightsensor.h"
```
```m
#include "yocto_lightsensor.h"
```
```pas
uses yocto_lightsensor;
```
```vb
yocto_lightsensor.vb
```
```cs
yocto_lightsensor.cs
```
```java
import com.yoctopuce.YoctoAPI.YLightSensor;
```
```py
from yocto_lightsensor import *
```

### Global functions

- **yFindLightSensor(func)**
  Retrieve 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()**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lightsensor-&gt;get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the light sensor in the form <code>SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;get_highestValue()</code></td>
<td>Returns the maximal value observed for the ambient light since the device was started.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;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>lightsensor-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the light sensor.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;get_lowestValue()</code></td>
<td>Returns the minimal value observed for the ambient light since the device was started.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;get_measureType()</code></td>
<td>Returns the type of light measure.</td>
</tr>
<tr>
<td><code>lightsensor-&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>lightsensor-&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>lightsensor-&gt;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>lightsensor-&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>lightsensor-&gt;get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;get_unit()</code></td>
<td>Returns the measuring unit for the ambient light.</td>
</tr>
<tr>
<td><code>lightsensor-&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>lightsensor-&gt;isOnline()</code></td>
<td>Checks if the light sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the light sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>lightsensor-&gt;load(msValidity)</code></td>
<td>Preloads the light sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the light sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>lightsensor-&gt;nextLightSensor()</code></td>
<td>Continues the enumeration of light sensors started using <code>yFirstLightSensor()</code>.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_highestValue(newval)</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the light sensor.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_measureType(newval)</code></td>
<td>Modify the light sensor type used in the device.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>lightsensor-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>lightsensor-&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>lightsensor-&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>
YLightSensor.FindLightSensor()  YLightSensor

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()

YLightSensor.FirstLightSensor()

YLightSensor.FirstLightSensor()

YLightSensor.FirstLightSensor()

YLightSensor.FirstLightSensor()

Starts the enumeration of light sensors currently accessible.

Use the method YLightSensor.nextLightSensor() to iterate on next light sensors.

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a pointer to a YLightSensor object, corresponding to the first light sensor currently online, or a null pointer if there are none.</td>
</tr>
</tbody>
</table>
### lightsensor\(\rightarrow\)calibrate() \(\rightarrow\) lightsensor.calibrate()

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```cpp
int calibrate( double calibratedVal)
```

**Parameters:**
- `calibratedVal` the desired target value.

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

On failure, throws an exception or returns a negative error code.
### Reference

**lightsensor.calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
YLightSensor

lightsensor.describe()

Returns a short text that describes unambiguously the instance of the light sensor in the form
TYPE(NAME)=SERIAL.FUNCTIONID.

String describe()

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the
function, SERIAL is the serial number of the module if the module is connected or "unresolved",
and FUNCTIONID is the hardware identifier of the function if the module is connected. For example,
this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the
module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module
has not yet been connected. This method does not trigger any USB or TCP transaction and can
therefore be used in a debugger.

Returns:
a string that describes the light sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
3. Reference

**YLightSensor**

<table>
<thead>
<tr>
<th>lightsensor -&gt; get_advertisedValue()</th>
<th>lightsensor -&gt; advertisedValue()</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightsensor.get_advertisedValue()</td>
<td></td>
</tr>
</tbody>
</table>

Returns the current value of the light sensor (no more than 6 characters).

**String get_advertisedValue()**

**Returns:**
- a string corresponding to the current value of the light sensor (no more than 6 characters).

On failure, throws an exception or returns **Y_ADVERTISEDVALUE_INVALID**.
YLightSensor

rightsensor→get_currentRawValue()
lightsensor→currentRawValue()
lightsensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in lux, 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 lux, as a floating point number

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
Returns the current value of the ambient light, in lux, as a floating point number.

```
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→get_errorMessage()
lightsensor→errorMessage()
lightsensor.get_errorMessage()

Returns the error message of the latest error with the light sensor.

String `get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:

- a string corresponding to the latest error message that occurred while using the light sensor object
YLightSensor

lightsensor.get_errorType()

Returns the numerical error code of the latest error with the light sensor.

int get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
a number corresponding to the code of the latest error that occurred while using the light sensor object
Retrieves a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

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`. 

```java
lightsensor.get_friendlyName()
```
**3. Reference**

```java
lightsensor.get_functionDescriptor()
```

**YLightSensor**

```java
lightsensor.functionDescriptor()
```

`lightsensor.get_functionDescriptor()`

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**String get_functionDescriptor()**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**

- an identifier of type `YFUN_DESCR`.

If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`. 
### 3. Reference

**lightsensor**→**get_functionId()**

**lightsensor**→**functionId()**

**lightsensor.get_functionId()**

Returns the hardware identifier of the light sensor, without reference to the module.

<table>
<thead>
<tr>
<th>String</th>
<th>get_functionId()</th>
</tr>
</thead>
</table>

For example, `relay1`

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string that identifies the light sensor (ex: <code>relay1</code>)</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
lightsensor→get_hardwareId()
lightsensor→hardwareId()
lightsensor.get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

String `get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the light sensor (for example `RELAYLO1-123456.relay1`).

Returns:
- a string that uniquely identifies the light sensor (ex: `RELAYLO1-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
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.
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 light sensor.

```java
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`. 
YLightSensor

`lightsensor.get_lowestValue()`

Returns the minimal value observed for the ambient light since the device was started.

```c
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`. 
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`. 
3. Reference

`lightsensor.get_module()`  
`lightsensor.module()`  
`lightsensor.get_module()`  

**gets the YModule object for the device on which the function is located.**

`YModule get_module()`

If the function cannot be located on any module, the returned instance of YModule is not shown as 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.
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String `get_reportFrequency()`

**Returns:**
- a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`. 
lightsensor->get_resolution()
lightsensor->resolution()
lightsensor.get_resolution()

Returns the resolution of the measured values.

double get_resolution()

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns:
- a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.
YLightSensor
lightsensor→get_unit()
lightsensor→unit()lightsensor.get_unit()

Returns the measuring unit for the ambient light.

String get_unit()

**Returns:**
- a string corresponding to the measuring unit for the ambient light

On failure, throws an exception or returns Y_UNIT_INVALID.
3. Reference

YLightSensor

`lightsensor→get_userData()`

`lightsensor→userData()`

`lightsensor.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.
YLightSensor

**Lightsensor.isOnline()**

Checks if the light sensor is currently reachable, without raising any error.

Iterate isOnline() ( )

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

**Returns:**

true if the light sensor can be reached, and false otherwise
lightsensor→load()\texttt{lightsensor.load()} \quad \textbf{YLightSensor}

Preloads the light sensor cache with a specified validity duration.

\textbf{int load( long msValidity)}

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

<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>\text{YAPI_SUCCESS}</td>
<td>when the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
lightsensor.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
YLightSensor\nextLightSensor()

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.
Reference

`lightsensor.registerTimedReportCallback()`

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( TimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.
lightsensor.registerValueCallback() YLightSensor

lightsensor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```cpp
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
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.
lightsensor→set_logicalName()
lightsensor→setLogicalName()
lightsensor.set_logicalName()

Changes the logical name of the light sensor.

```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 light 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 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.
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.
YLightSensor

`lightsensor->set_resolution()`

`lightsensor->setResolution()`

`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→set_userData()
lightsensor→setUserData()
lightsensor.set_userData()

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
3.24. 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:

```js
<script type='text/javascript' src='yocto_magnetometer.js'></script>
```
```
nodejs
var yoctolib = require('yoctolib');
var YMagnetometer = yoctolib.YMagnetometer;
```
```
php
require_once('yocto_magnetometer.php');
```
```
cpp
#include "yocto_magnetometer.h"
```
```
m
#include "yocto_magnetometer.h"
```
```
pas
uses yocto_magnetometer;
```
```
vb
yocto_magnetometer.vb
```
```
cs
yocto_magnetometer.cs
```
```
java
import com.yoctopuce.YoctoAPI.YMagnetometer;
```
```
py
from yocto_magnetometer import *
```

**Global functions**

- `yFindMagnetometer(func)`
  - Retrieves a magnetometer for a given identifier.

- `yFirstMagnetometer()`
  - Starts the enumeration of magnetometers currently accessible.

**YMagnetometer methods**

- `magnetometer→calibrateFromPoints(rawValues, refValues)`
  - Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

- `magnetometer→describe()`
  - Returns a short text that describes unambiguously the instance of the magnetometer in the form TYPE(NAME)=SERIAL.FUNCTIONID.

- `magnetometer→get_advertisedValue()`
  - Returns the current value of the magnetometer (no more than 6 characters).

- `magnetometer→get_currentRawValue()`
  - Returns the uncalibrated, unrounded raw value returned by the sensor, 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.
<table>
<thead>
<tr>
<th>Method Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_highestValue()</code></td>
<td>Returns the maximal value observed for the magnetic field 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 magnetometer.</td>
</tr>
<tr>
<td><code>get_lowestValue()</code></td>
<td>Returns the minimal value observed for the magnetic field 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 magnetic field.</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 magnetic field, as a floating point number.</td>
</tr>
<tr>
<td><code>get_yValue()</code></td>
<td>Returns the Y component of the magnetic field, as a floating point number.</td>
</tr>
<tr>
<td><code>get_zValue()</code></td>
<td>Returns the Z component of the magnetic field, as a floating point number.</td>
</tr>
<tr>
<td><code>isOnline()</code></td>
<td>Checks if the magnetometer is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>isOnline_async(callback, context)</code></td>
<td>Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>load(msValidity)</code></td>
<td>Preloads the magnetometer 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 magnetometer cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextMagnetometer()</code></td>
<td>Continues the enumeration of magnetometers started using <code>yFirstMagnetometer()</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>
</tbody>
</table>
3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>magnetometer.set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>magnetometer.set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>magnetometer.set_logicalName(newval)</code></td>
<td>Changes the logical name of the magnetometer.</td>
</tr>
<tr>
<td><code>magnetometer.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>magnetometer.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>magnetometer.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>magnetometer.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>magnetometer.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()

Retrieves a magnetometer for a given identifier.

YMagnetometer.FindMagnetometer( String func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method YMagnetometer.isOnline() to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters:

- func a string that uniquely characterizes the magnetometer

Returns:

- a YMagnetometer object allowing you to drive the magnetometer.
YMagnetometer.FirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

YMagnetometer FirstMagnetometer()

Use the method YMagnetometer.nextMagnetometer() to iterate on next magnetometers.

Returns:
- a pointer to a YMagnetometer object, corresponding to the first magnetometer currently online, or a null pointer if there are none.
magnetometer→calibrateFromPoints()  
magnetometer.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints( ArrayList<Double> rawValues,
                        ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

---

**Parameters**:
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns**:
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
magnetometer.describe() Returns a short text that describes unambiguously the instance of the magnetometer 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 magnetometer (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
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()
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.
magnetometer\rightarrow \textit{get\_currentValue()}
magnetometer\rightarrow \textit{currentValue()}
magnetometer.\textit{get\_currentValue()}

Returns the current value of the magnetic field, in mT, as a floating point number.

\textbf{double} \textit{get\_currentValue()}()

\textbf{Returns}:
  \begin{itemize}
  \item \textbf{a floating point number corresponding to the current value of the magnetic field, in mT, as a floating point number}
  \end{itemize}

\textbf{On failure, throws an exception or returns} \textbf{Y\_CURRENTVALUE\_INVALID}. 
3. Reference

YMagnetometer
magnetometer → get_errorMessage()
magnetometer → errorMessage()
magnetometer . get_errorMessage()

Returns the error message of the latest error with the magnetometer.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
a string corresponding to the latest error message that occurred while using the magnetometer object
 Returns the numerical error code of the latest error with the magnetometer.

```c
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**
a number corresponding to the code of the latest error that occurred while using the magnetometer object
3. Reference

YMagnetometer
magnetometer → get_friendlyName()
magnetometer → friendlyName()
magnetometer.get_friendlyName()

Returns a global identifier of the magnetometer in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName()

The returned string uses the logical names of the module and of the magnetometer if they are defined, otherwise the serial number of the module and the hardware identifier of the magnetometer (for example: MyCustomName.relay1)

Returns:

- a string that uniquely identifies the magnetometer using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
**magnetometer→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**String get_functionDescriptor()**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**

- an identifier of type `YFUN_DESCR`.

*If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.***
Returns the hardware identifier of the magnetometer, without reference to the module.

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**

- `get_hardwareId()`
- `hardwareId()`
- `magnetometer.get_hardwareId()`

Returns the unique hardware identifier of the magnetometer 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 magnetometer (for example `RELAYLO1-123456.relay1`).

**Returns:**
- `RELAYLO1-123456.relay1`

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.
<table>
<thead>
<tr>
<th>magnetometer (\rightarrow) get_highestValue()</th>
<th>YMagnetometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>magnetometer (\rightarrow) highestValue()</td>
<td></td>
</tr>
<tr>
<td>magnetometer (\rightarrow) get_highestValue()</td>
<td></td>
</tr>
</tbody>
</table>

Returns the maximal value observed for the magnetic field since the device was started.

```java
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.

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

**magnetometer→get_logicalName()**

**magnetometer→logicalName()**

**magnetometer.get_logicalName()**

<table>
<thead>
<tr>
<th>String <code>get_logicalName()</code></th>
</tr>
</thead>
</table>

**Returns**:

- a string corresponding to the logical name of the magnetometer.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
### YMagnetometer

<table>
<thead>
<tr>
<th>magnetometer → get_lowestValue()</th>
<th>magnetometer → lowestValue()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the minimal value observed for the magnetic field since the device was started.</td>
<td></td>
</tr>
</tbody>
</table>

```java
public double get_lowestValue() {
    // Implementation
}
```

**Returns:**

a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
magnetometer→get_module()          YMagnetometer
magnetometer→module()              
magnetometer.get_module()          

Gets the YModule object for the device on which the function is located.

YModule get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns:**

an instance of YModule
3. Reference

YMagnetometer

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.
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

`String get_reportFrequency()`

**Returns:**
- A string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`. 
magnetometer\rightarrow\text{get\_resolution()}
magnetometer\rightarrow\text{resolution()}
magnetometer.get\_resolution()

Returns the resolution of the measured values.

\text{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.

\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}.
YMagnetometer

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.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.
3. Reference

YMagnetometer

magnetometer→get_xValue() → magnetometer.xValue() → magnetometer.get_xValue()

Returns the X component of the magnetic field, as a floating point number.

`double get_xValue()`

**Returns:**
- a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns `Y_XVALUE_INVALID`. 
magnetometer→get_yValue()
magnetometer→yValue()
magnetometer.get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

double get_yValue()

Returns:
a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_YVALUE_INVALID.
### 3. Reference

<table>
<thead>
<tr>
<th>magnetometer → get_zValue()</th>
<th>YMagnetometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>magnetometer → zValue()</td>
<td></td>
</tr>
<tr>
<td>magnetometer.get_zValue()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a floating point number corresponding to the Z component of the magnetic field, as a floating point number</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_ZVALUE_INVALID`. 
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
Preloads the magnetometer cache with a specified validity duration.

```c
int load(long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
## YMagnetometer

**magnetometer.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

### Parameters :
- **rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- **refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

### Returns :
- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
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.

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()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
magnetometer→set_highestValue()  
magnetometer→setHighestValue()  
magnetometer.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.
magnetometer.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 magnetometer.

```
int set_logicalName( String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**
- `newval` a string corresponding to the logical name of the magnetometer.

**Returns :**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the recorded minimal value observed.

int set_lowestValue( double newval)

Parameters:
- newval a floating point number corresponding to the recorded minimal value observed

Returns:
- YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
magnetometer\rightarrow \text{set\_reportFrequency}()  
\text{YMagnetometer}  
magnetometer\rightarrow \text{setReportFrequency}()  
magnetometer.set\_reportFrequency()  

Changes the timed value notification frequency for this function.

```c
int \text{set\_reportFrequency}( \text{String } \text{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

YMagnetometer
magnetometer→setResolution()
magnetometer.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.
```plaintext
magnetometer.set_userData()
magnetometer->setUserData()
magnetometer.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

```void` set_userData(Object data)`

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters**:
- `data` any kind of object to be stored
3.25. 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;
```

```php
require_once('yocto_api.php');
```

```cpp
#include "yocto_api.h"
```

```m
#import "yocto_api.h"
```

```pas
uses yocto_api;
```

```vb
yocto_api.vb
```

```cs
yocto_api.cs
```

```java
import com.yoctopuce.YoctoAPI.YModule;
```

```py
from yocto_api import *
```

### YMeasure methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_averageValue()</code></td>
<td>Returns the average value observed during the time interval covered by this measure.</td>
</tr>
<tr>
<td><code>get_endTimeUTC()</code></td>
<td>Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).</td>
</tr>
<tr>
<td><code>get_maxValue()</code></td>
<td>Returns the largest value observed during the time interval covered by this measure.</td>
</tr>
<tr>
<td><code>get_minValue()</code></td>
<td>Returns the smallest value observed during the time interval covered by this measure.</td>
</tr>
<tr>
<td><code>get_startTimeUTC()</code></td>
<td>Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).</td>
</tr>
</tbody>
</table>
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.
3. Reference

YMeasure

measure\rightarrow get\_endTimeUTC() 
measure\rightarrow endTimeUTC() 
measure.get\_endTimeUTC() 

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

double get\_endTimeUTC() 

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

**Returns:**

- An floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.
Returns the largest value observed during the time interval covered by this measure.

```
double get_maxValue()
```

**Returns :**

a floating-point number corresponding to the largest value observed.
Returns the smallest value observed during the time interval covered by this measure.

```java
YMeasure

measure → get_minValue()
measure → minValue() measure.get_minValue()
```

**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 src='yocto_api.js'></script>
```
```
var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
```
```
require_once('yocto_api.php');
```
```
#include "yocto_api.h"
```
```
uses yocto_api;
```
```
function YAPI()
```
```
yocto_api.vb
```
```
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 \textit{r}th function on the module.

- **module→functionName(functionIndex)**
  
  Retrieves the logical name of the \textit{r}th function on the module.

- **module→functionValue(functionIndex)**
  
  Retrieves the advertised value of the \textit{r}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>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>module-&gt;get_hardwareId()</code></td>
<td>Returns the version of the firmware embedded in the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_icon2d()</code></td>
<td>Returns the unique hardware identifier of the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_lastLogs()</code></td>
<td>Returns the icon of the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_lastLogs()</code></td>
<td>Returns the unique hardware identifier of the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_lastLogs()</code></td>
<td>Returns the string with last logs of the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_luminosity()</code></td>
<td>Returns the luminosity of the module informative leds (from 0 to 100).</td>
</tr>
<tr>
<td><code>module-&gt;get_persistentSettings()</code></td>
<td>Returns the current state of persistent module settings.</td>
</tr>
<tr>
<td><code>module-&gt;get_productId()</code></td>
<td>Returns the USB device identifier of the module.</td>
</tr>
<tr>
<td><code>module-&gt;get_productName()</code></td>
<td>Returns the commercial name of the module, as set by the factory.</td>
</tr>
<tr>
<td><code>module-&gt;get_productRelease()</code></td>
<td>Returns the hardware release version of the module.</td>
</tr>
<tr>
<td><code>module-&gt;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>module-&gt;get_serialNumber()</code></td>
<td>Returns the serial number of the module, as set by the factory.</td>
</tr>
<tr>
<td><code>module-&gt;get_upTime()</code></td>
<td>Returns the number of milliseconds spent since the module was powered on.</td>
</tr>
<tr>
<td><code>module-&gt;get_usbCurrent()</code></td>
<td>Returns the current consumed by the module on the USB bus, in milli-amps.</td>
</tr>
<tr>
<td><code>module-&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>module-&gt;get_userVar()</code></td>
<td>Returns the value previously stored in this attribute.</td>
</tr>
<tr>
<td><code>module-&gt;isOnline()</code></td>
<td>Checks if the module is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>module-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the module is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>module-&gt;load(msValidity)</code></td>
<td>Preloads the module cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>module-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the module cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>module-&gt;nextModule()</code></td>
<td>Continues the module enumeration started using <code>yFirstModule()</code>.</td>
</tr>
<tr>
<td><code>module-&gt;reboot(secBeforeReboot)</code></td>
<td>Schedules a simple module reboot after the given number of seconds.</td>
</tr>
<tr>
<td><code>module-&gt;registerLogCallback(callback)</code></td>
<td>Registers a device log callback function.</td>
</tr>
</tbody>
</table>
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>module→revertFromFlash()</code></td>
<td>Reloads the settings stored in the nonvolatile memory, as when the module is powered on.</td>
</tr>
<tr>
<td><code>module→saveToFlash()</code></td>
<td>Saves current settings in the nonvolatile memory of the module.</td>
</tr>
<tr>
<td><code>module→set_allSettings(settings)</code></td>
<td>Restore all the setting of the module.</td>
</tr>
<tr>
<td><code>module→set_beacon(newval)</code></td>
<td>Turns on or off the module localization beacon.</td>
</tr>
<tr>
<td><code>module→set_logicalName(newval)</code></td>
<td>Changes the logical name of the module.</td>
</tr>
<tr>
<td><code>module→set_luminosity(newval)</code></td>
<td>Changes the luminosity of the module informative leds.</td>
</tr>
<tr>
<td><code>module→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>module→set_userVar(newval)</code></td>
<td>Returns the value previously stored in this attribute.</td>
</tr>
<tr>
<td><code>module→triggerFirmwareUpdate(secBeforeReboot)</code></td>
<td>Schedules a module reboot into special firmware update mode.</td>
</tr>
<tr>
<td><code>module→updateFirmware(path)</code></td>
<td>Prepare a firmware upgrade of the module.</td>
</tr>
<tr>
<td><code>module→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>
### YModule.FindModule()

**YModule.FindModule(String func)**

Allows you to find a module from its serial number or from its logical name.

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.
3. Reference

YModule.FirstModule()
yFirstModule() YModule.FirstModule()

Starts the enumeration of modules currently accessible.

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.
### checkFirmware()

Test if the byn file is valid for this module.

```java
String checkFirmware( String path, boolean 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>path</code></td>
<td>the path of a byn file or a directory that contain byn files</td>
</tr>
<tr>
<td><code>onlynew</code></td>
<td>return only files that are strictly newer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>the path of the byn file to use or a empty string if no byn files match the requirement</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a string that start with "error:".
module.describe() module.describe() YModule

Returns a descriptive text that identifies the module.

String describe()

The text may include either the logical name or the serial number of the module.

Returns:

- a string that describes the module
module\texttt{\rightarrow get\_beacon()} \quad \text{YModule}

\begin{tabular}{|p{0.9\textwidth}|}
\hline
\texttt{module\rightarrow beacon()} & module.\texttt{get\_beacon()} \\
\hline
Returns the state of the localization beacon. \\
\texttt{int get\_beacon()} &  \\
\hline
\begin{enumerate}
\item Returns : \\
\quad either \texttt{Y\_BEACON\_OFF} or \texttt{Y\_BEACON\_ON}, according to the state of the localization beacon \\
\item On failure, throws an exception or returns \texttt{Y\_BEACON\_INVALID}. \\
\end{enumerate}
\end{tabular}
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>module.get_errorMessage()</code></td>
<td>Returns the error message of the latest error with this module object.</td>
</tr>
<tr>
<td><code>String get_errorMessage()</code></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 this module object.
Retrieves the numerical error code of the latest error with this module object.

```c
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

A number corresponding to the code of the latest error that occurred while using this module object.
Returns the version of the firmware embedded in the module.

String `get_firmwareRelease()`

Returns:
- a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`. 
module → get_hardwareId()
module → hardwareId() module.get_hardwareId()

YModule

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
3. Reference

**YModule**

**module**→**get_lastLogs()**

**module**→**lastLogs()**

```
module.get_lastLogs()
```

Returns a string with last logs of the module.

**String get_lastLogs()**

This method return only logs that are still in the module.

**Returns:**

a string with last logs of the module. On failure, throws an exception or returns YAPI_INVALID_STRING.
**module**→**get_logicalName()**

**Returns the logical name of the module.**

String **get_logicalName()**

**Returns:**

a string corresponding to the logical name of the module

On failure, throws an exception or returns **Y_LOGICALNAME_INVALID**.
module \rightarrow \text{get\_luminosity()} 
module \rightarrow \text{luminosity()} \text{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 `Y\_LUMINOSITY\_INVALID`. 
module\rightarrow get\_persistentSettings() 
module\rightarrow persistentSettings() 
module.get_persistentSettings() 

Returns the current state of persistent module settings.

int get_persistentSettings( )

Returns :

a value among Y\_PERSISTENTSETTINGS\_LOADED, Y\_PERSISTENTSETTINGS\_SAVED and Y\_PERSISTENTSETTINGS\_MODIFIED corresponding to the current state of persistent module settings

On failure, throws an exception or returns Y\_PERSISTENTSETTINGS\_INVALID.
Returns the USB device identifier of the module.

\[
\text{int } \text{get_productId}() \quad \text{YModule}
\]

\[
\text{module} \rightarrow \text{get_productId}() \quad \text{module} \rightarrow \text{productId}() \quad \text{module.get_productId}()
\]

Returns:
- an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y_PRODUCTID_INVALID.
YModule

module→get_productName()
module→productName()
module.get_productName()

Returns the commercial name of the module, as set by the factory.

String get_productName()

Returns:
- a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y_PRODUCTNAME_INVALID.
YModule

module → get_productRelease()
module → productRelease()
module.get_productRelease()

Returns the hardware release version of the module.

int get_productRelease()

Returns:
- an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.
YModule

module→get_rebootCountdown()
module→rebootCountdown()
module.get_rebootCountdown()  

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

int get_rebootCountdown()  

Returns:

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns Y_REBOOTCOUNTROUND_INVALID.
Returns the serial number of the module, as set by the factory.

String get_serialNumber()

Returns:
- a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns Y_SERIALNUMBER_INVALID.
Returns the number of milliseconds spent since the module was powered on.

Returns:

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns Y_UPTIME_INVALID.
module.get_usbCurrent() Returns the current consumed by the module on the USB bus, in milli-amps.

int get_usbCurrent()

Returns:
- an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns Y_USBCURRENT_INVALID.
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 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()**

Checks if the module is currently reachable, without raising any error.

```java
boolean isOnline()
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

**Returns:**
- `true` if the module can be reached, and `false` otherwise
### module\rightarrow{load()} \texttt{module.load()} \texttt{YModule}

Preloads the module cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

<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 module 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.
YModule

nextModule() module

Continues the module enumeration started using \texttt{yFirstModule()}.

\textbf{YModule} \texttt{nextModule()} 

\textbf{Returns :}

a pointer to a \texttt{YModule} object, corresponding to the next module found, or a null pointer if there are no more modules to enumerate.
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.
Registers a device log callback function.

```c
void 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, and
  - the character string containing the log.
module→revertFromFlash()  
module.revertFromFlash()

Reloads 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.
3. Reference

YModule

module.saveToFlash() saves current settings in the nonvolatile memory of the module.

int saveToFlash()

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns:

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.
module->set_allSettings()  \(\text{YModule}\)
module->setAllSettings()
module.set_allSettings()

Restore all the setting of the module.

```c
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.
YModule

module→set_beacon()
module→setBeacon() module.set_beacon()

Turns on or off the module localization beacon.

int set_beacon( int newval)

Parameters :
newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :
YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the logical name of the module.

### 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()
module.setUserData()module.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

`void set_userData( Object data)`

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters**:
- `data` any kind of object to be stored
Returns the value previously stored in this attribute.

**Parameters:**

- `newval` an integer

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
### YModule

#### module.triggerFirmwareUpdate()

Schedules a module reboot into special firmware update mode.

```c
int triggerFirmwareUpdate(int secBeforeReboot)
```

**Parameters:**
- `secBeforeReboot` number of seconds before rebooting

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
YModule

module.updateFirmware()

Prepare a firmware upgrade of the module.

YFirmwareUpdate updateFirmware(String path)

This method returns an object `YFirmwareUpdate` which will handle the firmware upgrade process.

**Parameters:**
- `path` the path of the byn file to use.

**Returns:**
- A `YFirmwareUpdate` object.
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"
```
```m
uses yocto_motor;
```
```vb
yocto_motor.vb
```
```cs
yocto_motor.cs
```
```java
import com.yoctopuce.YoctoAPI.YMotor;
```
```py
from yocto_motor import *
```

### Global functions

**yFindMotor(func)**

Retrieves a motor for a given identifier.

**yFirstMotor()**

Starts the enumeration of motors currently accessible.

### YMотор methods

**motor→brakingForceMove(targetPower, delay)**

Changes progressively the braking force applied to the motor for a specific duration.

**motor→describe()**

Returns a short text that describes unambiguously the instance of the motor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**motor→drivingForceMove(targetPower, delay)**

Changes progressively the power sent to the motor for a specific duration.

**motor→get_advertisedValue()**

Returns the current value of the motor (no more than 6 characters).

**motor→get_brakingForce()**

Returns the braking force applied to the motor, as a percentage.

**motor→get_cutoFFVoltage()**

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

**motor→get_drivingForce()**

Returns the power sent to the motor, as a percentage between -100% and +100%.

**motor→get_errorMessage()**

Returns the error message of the latest error with the motor.

**motor→get_errorType()**

Returns the numerical error code of the latest error with the motor.

**motor→get_failSafeTimeout()**

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>motor→get_frequency()</code></td>
<td>Returns the PWM frequency used to control the motor.</td>
</tr>
<tr>
<td><code>motor→get_friendlyName()</code></td>
<td>Returns a global identifier of the motor in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><code>motor→get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><code>motor→get_functionId()</code></td>
<td>Returns the hardware identifier of the motor, without reference to the module.</td>
</tr>
<tr>
<td><code>motor→get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the motor in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>motor→get_logicalName()</code></td>
<td>Returns the logical name of the motor.</td>
</tr>
<tr>
<td><code>motor→get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>motor→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>motor→get_motorStatus()</code></td>
<td>Return the controller state.</td>
</tr>
<tr>
<td><code>motor→get_overCurrentLimit()</code></td>
<td>Returns the current threshold (in mA) above which the controller automatically switches to error state.</td>
</tr>
<tr>
<td><code>motor→get_starterTime()</code></td>
<td>Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.</td>
</tr>
<tr>
<td><code>motor→get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><code>motor→isOnline()</code></td>
<td>Checks if the motor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>motor→isOnline_async(callback, context)</code></td>
<td>Checks if the motor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>motor→keepALive()</code></td>
<td>Rearms the controller failsafe timer.</td>
</tr>
<tr>
<td><code>motor→load(msValidity)</code></td>
<td>Preloads the motor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>motor→load_async(msValidity, callback, context)</code></td>
<td>Preloads the motor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>motor→nextMotor()</code></td>
<td>Continues the enumeration of motors started using yFirstMotor().</td>
</tr>
<tr>
<td><code>motor→registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>motor→resetStatus()</code></td>
<td>Reset the controller state to IDLE.</td>
</tr>
<tr>
<td><code>motor→set_brakingForce(newval)</code></td>
<td>Changes immediately the braking force applied to the motor (in percents).</td>
</tr>
<tr>
<td><code>motor→set_cutOffVoltage(newval)</code></td>
<td>Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.</td>
</tr>
</tbody>
</table>
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>motor.set_drivingForce(newval)</code></td>
<td>Changes immediately the power sent to the motor.</td>
</tr>
<tr>
<td><code>motor.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.set_frequency(newval)</code></td>
<td>Changes the PWM frequency used to control the motor.</td>
</tr>
<tr>
<td><code>motor.set_logicalName(newval)</code></td>
<td>Changes the logical name of the motor.</td>
</tr>
<tr>
<td><code>motor.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.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.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.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()

YMotor.FindMotor(String func)

Retrieves a motor 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 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.
3. Reference

**YMotor.FirstMotor()**

*yFirstMotor() YMotor.FirstMotor()

Starts the enumeration of motors currently accessible.

---

**YMotor FirstMotor()**

Use the method `YMotor.nextMotor()` to iterate on next motors.

**Returns:**

- a pointer to a `YMotor` object, corresponding to the first motor currently online, or a null pointer if there are none.
**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.
### 3. Reference

**Motor**

Returns a short text that describes unambiguously the instance of the motor 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(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`)

---

**YMotor**

Returns a short text that describes unambiguously the instance of the motor 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(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`)

---
YMotor

motor→drivingForceMove()
motor.drivingForceMove()

Changes progressively the power sent to the moteur for a specific duration.

```
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.
3. Reference

```java
YMotor

motor.get_advertisedValue()
motor.advertisedValue()
motor.get_advertisedValue()
```

Returns the current value of the motor (no more than 6 characters).

```java
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`. 
YMotor

\[
\text{motor} \rightarrow \text{get\_brakingForce()}
\]

\[
\text{motor} \rightarrow \text{brakingForce()}
\]

\[
\text{motor}.\text{get\_brakingForce()}
\]

Returns the braking force applied to the motor, as a percentage.

\[
\text{double get\_brakingForce()}
\]

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 \text{Y\_BRAKINGFORCE\_INVALID}.
motor→get_cutOffVoltage()
motor→cutOffVoltage()
motor.get_cutOffVoltage()

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

double get_cutOffVoltage( )

This setting prevents damage to a battery that can occur when drawing current from an "empty" battery.

Returns:
- a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

On failure, throws an exception or returns Y_CUTOFFVOLTAGE_INVALID.
Returns the power sent to the motor, as a percentage between -100% and +100%.

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.
3. Reference

**YMotor**

`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
>Returns the numerical error code of the latest error with the motor.

    int get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

    Returns :
    a number corresponding to the code of the latest error that occurred while using the motor object
3. Reference

```c
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`. 
motor→get_frequency()
motor→frequency()motor.get_frequency()

Returns the PWM frequency used to control the motor.

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.
### Reference

YMotor

<table>
<thead>
<tr>
<th>motor→get_friendlyName()</th>
<th>YMotor</th>
</tr>
</thead>
<tbody>
<tr>
<td>motor→friendlyName()</td>
<td>motor.get_friendlyName()</td>
</tr>
</tbody>
</table>

Returns a global identifier of the motor in the format `MODULE_NAME.FUNCTION_NAME`.

```java
String get_friendlyName() { }
```

The returned string uses the logical names of the module and of the motor if they are defined, otherwise the serial number of the module and the hardware identifier of the motor (for example: `MyCustomName.relay1`)

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string that uniquely identifies the motor using logical names (ex: <code>MyCustomName.relay1</code>)</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.</td>
</tr>
<tr>
<td></td>
<td>This identifier can be used to test if two instances of <code>YFunction</code> reference the same physical function on the same physical device.</td>
</tr>
<tr>
<td><strong>Returns:</strong></td>
<td></td>
</tr>
<tr>
<td>an identifier of type <code>YFUN_DESCR</code>.</td>
<td></td>
</tr>
<tr>
<td><strong>If the function has never been contacted, the returned value is</strong></td>
<td><strong>Y_FUNCTIONDESCRIPTOR_INVALID.</strong></td>
</tr>
</tbody>
</table>
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 (e.g. `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
<table>
<thead>
<tr>
<th>YMotor</th>
</tr>
</thead>
</table>
| **motor**→**get_hardwareId()** | Returns the unique hardware identifier of the motor in the form **SERIAL.FUNCTIONID**.  
| **motor**→**hardwareId()** |  
| **motor.get_hardwareId()** | Returns the unique hardware identifier of the motor 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 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**.
3. Reference

### YMotor

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.
YMotor

motor\rightarrow get\_module()
motor\rightarrow module() motor.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>YModule get_module( )</td>
</tr>
</tbody>
</table>

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

Return the controller state.

```c
int get_motorStatus() {
    // Possible states...
    //...
    // Returns:
    // a value among Y_MOTORSTATUS_IDLE, Y_MOTORSTATUS_BRAKE,
    // Y_MOTORSTATUS_FORWD, Y_MOTORSTATUS_BACKWD, Y_MOTORSTATUS_LOVOLT,
    // Y_MOTORSTATUS_HICURR, Y_MOTORSTATUS_HIHEAT and Y_MOTORSTATUS_FAILSF
}
```

On failure, throws an exception or returns Y_MOTORSTATUS_INVALID.
motor→get_overCurrentLimit()
motor→overCurrentLimit()
motor.get_overCurrentLimit()

Returns the current threshold (in mA) above which the controller automatically switches to error state.

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 **Y_OVERCURRENTLIMIT_INVALID**.
### Reference

`motor->get_starterTime()`  
`motor->starterTime()`  
`motor.get_starterTime()`  

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

```cpp
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`. 
**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 motor is currently reachable, without raising any error.

If there is a cached value for the motor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the motor.

**Returns:**
- true if the motor can be reached, and false otherwise
`motor→keepALive()` \textit{motor.\texttt{keepALive}()} \hfill \textit{YMotor}

Rearms the controller failsafe timer.

```c
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 \textit{set} function implicitly rearms the failsafe timer.
Preloads the motor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- **msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
**YMotor**

**nextMotor()**

Continues the enumeration of motors started using *yFirstMotor()*.

**YMotor nextMotor()**

**Returns:**

A pointer to a **YMotor** object, corresponding to a motor currently online, or a *null* pointer if there are no more motors to enumerate.
motor.registerValueCallback()  

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
motor → resetStatus() \[ \text{motor.resetStatus()} \]

Reset the controller state to IDLE.

```c
int resetStatus()
```

This function must be invoked explicitly after any error condition is signaled.
Changes immediately the braking force applied to the motor (in percents).

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.
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.
Changes immediately the power sent to the motor.

```c
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.
motor.set_failSafeTimeout()

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

```
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.
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.
### Reference

```cpp
YMotor
motor.set_overCurrentLimit()
```

Changes the current threshold (in mA) above which the controller automatically switches to error state.

```cpp
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.
motor\rightarrow\text{set\_starterTime()}
motor\rightarrow\text{setStarterTime()}
motor.set\_starterTime()

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

\text{int} \ \text{set\_starterTime( int newval)}

\textbf{Parameters :}

- \texttt{newval} an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

\textbf{Returns :}

- \texttt{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 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
3.28. 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
#import "yocto_network.h"
```
```pas
uses yocto_network;
```
```vb
yocto_network.vb
```
```cs
yocto_network.cs
```
```java
import com.yoctopuce.YoctoAPI.YNetwork;
```
```py
from yocto_network import *
```

---

### Global functions

- **yFindNetwork(func)**
  - Retrieves a network interface for a given identifier.
- **yFirstNetwork()**
  - Starts the enumeration of network interfaces currently accessible.

### YNetwork methods

- **network→callbackLogin(username, password)**
  - Connects to the notification callback and saves the credentials required to log into it.
- **network→describe()**
  - Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
- **network→get_advertisedValue()**
  - Returns the current value of the network interface (no more than 6 characters).
- **network→get_adminPassword()**
  - Returns a hash string if a password has been set for user "admin", or an empty string otherwise.
- **network→get_callbackCredentials()**
  - Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.
- **network→get_callbackEncoding()**
  - Returns the encoding standard to use for representing notification values.
- **network→get_callbackMaxDelay()**
  - Returns the maximum waiting time between two callback notifications, in seconds.
- **network→get_callbackMethod()**
  - Returns the HTTP method used to notify callbacks for significant state changes.
- **network→get_callbackMinDelay()**
  - Returns the minimum waiting time between two callback notifications, in seconds.
- **network→get_callbackUrl()**
  - Returns the callback URL to notify of significant state changes.
- **network→get_discoverable()**
  - Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>network-&gt;get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the network interface.</td>
</tr>
<tr>
<td><code>network-&gt;get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the network interface.</td>
</tr>
<tr>
<td><code>network-&gt;get_friendlyName()</code></td>
<td>Returns a global identifier of the network interface in the format <code>MODULE_NAME.FUNCTION_NAME</code>.</td>
</tr>
<tr>
<td><code>network-&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>network-&gt;get_functionId()</code></td>
<td>Returns the hardware identifier of the network interface, without reference to the module.</td>
</tr>
<tr>
<td><code>network-&gt;get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the network interface in the form <code>SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>network-&gt;get_ipAddress()</code></td>
<td>Returns the IP address currently in use by the device.</td>
</tr>
<tr>
<td><code>network-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the network interface.</td>
</tr>
<tr>
<td><code>network-&gt;get_macAddress()</code></td>
<td>Returns the MAC address of the network interface.</td>
</tr>
<tr>
<td><code>network-&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>network-&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>network-&gt;get_poeCurrent()</code></td>
<td>Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.</td>
</tr>
<tr>
<td><code>network-&gt;get_primaryDNS()</code></td>
<td>Returns the IP address of the primary name server to be used by the module.</td>
</tr>
<tr>
<td><code>network-&gt;get_readiness()</code></td>
<td>Returns the current established working mode of the network interface.</td>
</tr>
<tr>
<td><code>network-&gt;get_router()</code></td>
<td>Returns the IP address of the router on the device subnet (default gateway).</td>
</tr>
<tr>
<td><code>network-&gt;get_secondaryDNS()</code></td>
<td>Returns the IP address of the secondary name server to be used by the module.</td>
</tr>
<tr>
<td><code>network-&gt;get_subnetMask()</code></td>
<td>Returns the subnet mask currently used by the device.</td>
</tr>
<tr>
<td><code>network-&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>network-&gt;get_userPassword()</code></td>
<td>Returns a hash string if a password has been set for &quot;user&quot; user, or an empty string otherwise.</td>
</tr>
<tr>
<td><code>network-&gt;get_wwwWatchdogDelay()</code></td>
<td>Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.</td>
</tr>
<tr>
<td><code>network-&gt;isOnline()</code></td>
<td>Checks if the network interface is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>network-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the network interface is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>network-&gt;load(msValidity)</code></td>
<td>Preloads the network interface cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>network-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the network interface cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>network-&gt;nextNetwork()</code></td>
<td>Continues the enumeration of network interfaces started using <code>yFirstNetwork()</code>.</td>
</tr>
<tr>
<td><code>network-&gt;ping(host)</code></td>
<td>Pings <code>str_host</code> to test the network connectivity.</td>
</tr>
<tr>
<td><code>network-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>network-&gt;set_adminPassword(newval)</code></td>
<td>Changes the password for the “admin” user.</td>
</tr>
<tr>
<td><code>network-&gt;set_callbackCredentials(newval)</code></td>
<td>Changes the credentials required to connect to the callback address.</td>
</tr>
<tr>
<td><code>network-&gt;set_callbackEncoding(newval)</code></td>
<td>Changes the encoding standard to use for representing notification values.</td>
</tr>
<tr>
<td><code>network-&gt;set_callbackMaxDelay(newval)</code></td>
<td>Changes the maximum waiting time between two callback notifications, in seconds.</td>
</tr>
<tr>
<td><code>network-&gt;set_callbackMethod(newval)</code></td>
<td>Changes the HTTP method used to notify callbacks for significant state changes.</td>
</tr>
<tr>
<td><code>network-&gt;set_callbackMinDelay(newval)</code></td>
<td>Changes the minimum waiting time between two callback notifications, in seconds.</td>
</tr>
<tr>
<td><code>network-&gt;set_callbackUrl(newval)</code></td>
<td>Changes the callback URL to notify significant state changes.</td>
</tr>
<tr>
<td><code>network-&gt;set_discoverable(newval)</code></td>
<td>Changes 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><code>network-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the network interface.</td>
</tr>
<tr>
<td><code>network-&gt;set_primaryDNS(newval)</code></td>
<td>Changes the IP address of the primary name server to be used by the module.</td>
</tr>
<tr>
<td><code>network-&gt;set_secondaryDNS(newval)</code></td>
<td>Changes the IP address of the secondary name server to be used by the module.</td>
</tr>
<tr>
<td><code>network-&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>network-&gt;set_userPassword(newval)</code></td>
<td>Changes the password for the “user” user.</td>
</tr>
<tr>
<td><code>network-&gt;set_wwwWatchdogDelay(newval)</code></td>
<td>Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.</td>
</tr>
<tr>
<td><code>network-&gt;useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)</code></td>
<td>Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.</td>
</tr>
<tr>
<td><code>network-&gt;useStaticIP(ipAddress, subnetMaskLen, router)</code></td>
<td>Changes the configuration of the network interface to use a static IP address.</td>
</tr>
<tr>
<td><code>network-&gt;wait_async(callback, context)</code></td>
<td></td>
</tr>
</tbody>
</table>
3. Reference

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YNetwork.FindNetwork()

YNetwork.FindNetwork()

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()

Starts the enumeration of network interfaces currently accessible.

Use the method YNetwork.nextNetwork() to iterate on next network interfaces.

Returns:
- a pointer to a YNetwork object, corresponding to the first network interface currently online, or a null pointer if there are none.
**network.callbackLogin()**

Connects to the notification callback and saves the credentials required to log into it.

```java
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.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>username</strong></td>
</tr>
<tr>
<td><strong>password</strong></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.
Returns a short text that describes unambiguously the instance of the network interface in the form
\text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID}.

More precisely, \text{TYPE} is the type of the function, \text{NAME} its 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(BadCustomerName.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: \text{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1})
network→get_adminPassword()
network→adminPassword()
network.get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

String get_adminPassword()  

Returns:
  a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns Y_ADMINPASSWORD_INVALID.
network→get_advertisedValue()
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.
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.
```
Returns the maximum waiting time between two callback notifications, in seconds.

```c
int get_callbackMaxDelay()
```

**Returns:**

an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMAXDELAY_INVALID.
Returns the HTTP method used to notify callbacks for significant state changes.

Returns:

- A value among Y_CALLBACKMETHOD_POST, Y_CALLBACKMETHOD_GET, and Y_CALLBACKMETHOD_PUT corresponding to the HTTP method used to notify callbacks for significant state changes.

On failure, throws an exception or returns Y_CALLBACKMETHOD_INVALID.
network→get_callbackMinDelay()  
YNetwork
network→callbackMinDelay()  
network.get_callbackMinDelay()  

Returns the minimum waiting time between two callback notifications, in seconds.

int get_callbackMinDelay()  

Returns:

- an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns \texttt{Y\_CALLBACKMINDELAY\_INVALID}.
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).

**Returns:**

- either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns `Y_DISCOVERABLE_INVALID`. 

```c
int get_discoverable( )
```
network\to get\_errorMessage()

network\to errorMessage()

\texttt{network.get\_errorMessage()}\hspace{1cm} \texttt{YNetwork}

Returns the error message of the latest error with the network interface.

\textbf{String get\_errorMessage()}\hspace{1cm} \texttt{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 network interface object
network \rightarrow \text{get\_errorType}() \\
\text{network} \rightarrow \text{errorType()} \text{network} . \text{get\_errorType}() \\

Returns the numerical error code of the latest error with the network interface.

\text{int \text{get\_errorType}()} \\

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

\textbf{Returns:} \\
\text{a number corresponding to the code of the latest error that occurred while using the network interface object}
**get_friendlyName()**

YNetwork

```java
network.get_friendlyName()
```

**friendlyName()**

```java
network.friendlyName()
```

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

String `get_friendlyName()`

The returned string uses the logical names of the module and of the network interface if they are defined, otherwise the serial number of the module and the hardware identifier of the network interface (for 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()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

- an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTION_INVALID.
YNetwork

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()  \hspace{1cm} \text{YNetwork}

network->hardwareId() \hspace{1cm} \text{network.get_hardwareId()}

Returns the unique hardware identifier of the network interface in the form \text{SERIAL.FUNCTIONID}.

\textbf{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 \text{RELAYLO1-123456.relay1}).

\textbf{Returns :}

a string that uniquely identifies the network interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns \text{Y_HARDWAREID_INVALID}. 
network → get_ipAddress()

YNetwork

network → ipAddress() network.get_ipAddress()

Returns the IP address currently in use by the device.

String get_ipAddress()

The address may have been configured statically, or provided by a DHCP server.

**Returns:**

- a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns Y_IPADDRESS_INVALID.
Returns the logical name of the network interface.

String **get_logicalName()**

**Returns:**

a string corresponding to the logical name of the network interface.

On failure, throws an exception or returns **Y_LOGICALNAME_INVALID**.
network\rightarrow \textbf{get\_macAddress()} \quad \text{YNetwork}

network\rightarrow \textbf{macAddress()}

\texttt{network\.get\_macAddress()} \quad \text{String \textbf{get\_macAddress()}}

Returns the MAC address of the network interface.

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 \texttt{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
3. Reference

network\rightarrow get\_poeCurrent() \hspace{1cm} YNetwork

network\rightarrow poeCurrent() \hspace{1cm} network.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() {
    // Implementation
}
```

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.
<table>
<thead>
<tr>
<th>method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>network.get_router()</code></td>
<td>Returns the IP address of the router on the device subnet (default gateway).</td>
</tr>
<tr>
<td><code>String get_router()</code></td>
<td></td>
</tr>
<tr>
<td><strong>Returns</strong>:</td>
<td>a string corresponding to the IP address of the router on the device subnet (default gateway)</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns <code>Y_ROUTER_INVALID</code>.</td>
</tr>
</tbody>
</table>
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.
network → get_userData() → userData() → network.get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

Object get_userData()

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns:
the object stored previously by the caller.
Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

Returns:
- a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise.

On failure, throws an exception or returns Y_USERPASSWORD_INVALID.
network→get_wwwWatchdogDelay()  YNetwork
network→wwwWatchdogDelay()
network.get_wwwWatchdogDelay()  Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

int get_wwwWatchdogDelay()  A zero value disables automated reboot in case of Internet connectivity loss.

Returns:
- an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

On failure, throws an exception or returns Y_WWWWATCHDOGDELAY_INVALID.
Checks if the network interface is currently reachable, without raising any error.

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

**Returns:**

- `true` if the network interface can be reached, and `false` otherwise
network→load()network.load()

Preloads the network interface cache with a specified validity duration.

`int load( long msValidity)`

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
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.
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\rightarrow\text{registerValueCallback()}

network.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{int registerValueCallback( UpdateCallback \textbf{callback})} \\
\hline
\end{tabular}
\end{center}

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{center}
\textbf{Parameters :}
\begin{itemize}
    \item \textbf{callback} the callback 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}
\end{center}
3. Reference

Changes the password for the "admin" user.

```c
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.

```java
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_YOCTO_API` corresponding to the encoding standard to use for representing notification values.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the maximum waiting time between two callback notifications, in seconds.

\[
\text{int } \text{set_callbackMaxDelay}( \text{int } \text{newval})
\]

**Parameters:**
- \text{newval} an integer corresponding to the maximum waiting time between two callback notifications, in seconds

**Returns:**
- \text{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.
YNetwork

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.
network→set_logicalName()
network→setLogicalName()
network.set_logicalName()

Changes the logical name of the network interface.

```java
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.

```c
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.

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.
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
network→set_userPassword()
network→setUserPassword()
network.set_userPassword()

Changes the password for the "user" user.

int set_userPassword( String newval)

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the saveToFlash() method of the module if the modification must be kept.

Parameters :
   newval a string corresponding to the password for the "user" user

Returns :
   YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
network.set_wwwWatchdogDelay()

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.

```java
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 (e.g., 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 OS control 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 the -o option.

In order to use the functions described here, you should include:

```js
<script type='text/javascript' src='yocto_oscontrol.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YOsControl = yoctolib.YOsControl;
```
```php
require_once('yocto_oscontrol.php');
```
```cpp
#include "yocto_oscontrol.h"
```
```m
#import "yocto_oscontrol.h"
```
```pas
uses yocto_oscontrol;
```
```vb
yocto_oscontrol.vb
```
```cs
yocto_oscontrol.cs
```
```java
import com.yoctopuce.YoctoAPI.YOsControl;
```
```py
from yocto_oscontrol import *
```

Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindOsControl(func)</code></td>
<td>Retrieves OS control for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstOsControl()</code></td>
<td>Starts the enumeration of OS control currently accessible.</td>
</tr>
</tbody>
</table>

YOsControl methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oscontrol→describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the OS control in the form TYPE(Name)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>oscontrol→get_advertisedValue()</code></td>
<td>Returns the current value of the OS control (no more than 6 characters).</td>
</tr>
<tr>
<td><code>oscontrol→get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the OS control.</td>
</tr>
<tr>
<td><code>oscontrol→get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the OS control.</td>
</tr>
<tr>
<td><code>oscontrol→get_friendlyName()</code></td>
<td>Returns a global identifier of the OS control in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><code>oscontrol→get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><code>oscontrol→get_functionId()</code></td>
<td>Returns the hardware identifier of the OS control, without reference to the module.</td>
</tr>
<tr>
<td><code>oscontrol→get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the OS control in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>oscontrol→get_logicalName()</code></td>
<td>Returns the logical name of the OS control.</td>
</tr>
<tr>
<td><code>oscontrol→get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>oscontrol→get_module_async(callback, context)</code></td>
<td></td>
</tr>
<tr>
<td>Method Call</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>oscontrol-&gt;get_shutdownCountdown()</code></td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version). Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.</td>
</tr>
<tr>
<td><code>oscontrol-&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>oscontrol-&gt;isOnline()</code></td>
<td>Checks if the OS control is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>oscontrol-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the OS control is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>oscontrol-&gt;load(msValidity)</code></td>
<td>Preloads the OS control cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>oscontrol-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the OS control cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>oscontrol-&gt;nextOsControl()</code></td>
<td>Continues the enumeration of OS control started using <code>yFirstOsControl()</code>.</td>
</tr>
<tr>
<td><code>oscontrol-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>oscontrol-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the OS control.</td>
</tr>
<tr>
<td><code>oscontrol-&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>oscontrol-&gt;shutdown(secBeforeShutDown)</code></td>
<td>Schedules an OS shutdown after a given number of seconds.</td>
</tr>
<tr>
<td><code>oscontrol-&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

**YOsControl.FindOsControl()**

Retrieves OS control for a given identifier.

**YOsControl.FindOsControl( String func)**

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method **YOsControl.isOnline()** to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**
- **func** a string that uniquely characterizes the OS control

**Returns :**
- a **YOsControl** object allowing you to drive the OS control.
YOsControl.FirstOsControl()

**YOsControl.FirstOsControl()**

Starts the enumeration of OS control currently accessible.

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

**Returns:**

A pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a null pointer if there are none.
Returns a short text that describes unambiguously the instance of the OS control in the form
\texttt{TYPE(NAME)=SERIAL.FUNCTIONID}.

\textbf{String \texttt{describe( )}}

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.

\textbf{Returns :}

\begin{itemize}
  \item a string that describes the OS control (ex: \texttt{Relay(MyCustomName.relay1)=RELAYLO1-}
  \texttt{123456.relay1})
\end{itemize}
| oscontrol→get_advertisedValue() | YOsControl |
| oscontrol→advertisedValue()   |             |
| oscontrol.get_advertisedValue() |             |

Returns the current value of the OS control (no more than 6 characters).

```java
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.
}
```
3. Reference

YOsControl

oscontrol→get_errorMessage()
oscontrol→errorMessage()
oscontrol.get_errorMessage()

Returns the error message of the latest error with the OS control.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
- a string corresponding to the latest error message that occurred while using the OS control object
YOsControl

```
FUNCTION
oscontrol\to get\_errorType()
oscontrol\to errorType()
oscontrol.\text{get\_errorType}()
```

Returns the numerical error code of the latest error with the OS control.

```c
int get\_errorType()
```

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 number corresponding to the code of the latest error that occurred while using the OS control object</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td><code>oscontrol.get_friendlyName()</code></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Returns</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
YOsControl

oscontrol→get_functionDescriptor()
oscontrol→functionDescriptor()
oscontrol.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor() 

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_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.

```java
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.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the OS control (for example RELAY01-123456.relay1).

Returns:
- a string that uniquely identifies the OS control (ex: RELAY01-123456.relay1)
- On failure, throws an exception or returns Y_HARDWAREID_INVALID.
<table>
<thead>
<tr>
<th>Returns:</th>
<th>a string corresponding to the logical name of the OS control.</th>
</tr>
</thead>
<tbody>
<tr>
<td>On failure, throws an exception or returns</td>
<td><em>Y_LOGICALNAME_INVALID</em>.</td>
</tr>
</tbody>
</table>
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.
oscontrol→get_shutdownCountdown()

oscontrol→shutdownCountdown()

oscontrol.get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

```c
int get_shutdownCountdown()
```

Returns:
- an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

On failure, throws an exception or returns Y_SHUTDOWNCOUNTDOWN_INVALID.
oscontrol→getUserData()
oscontrol→userData(oscontrol.getUserData())

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.
<table>
<thead>
<tr>
<th>oscontrol→isOnline()</th>
<th>oscontrol.isOnline()</th>
<th>YOsControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checks if the OS control is currently reachable, without raising any error.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```java
boolean isOnline()
```

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

**Returns:**
- **true** if the OS control can be reached, and **false** otherwise.
**oscontrol→load() oscontrol.load()**

Preloads the OS control cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- **msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
oscontrol->nextOsControl() YOsControl

Continues the enumeration of OS control started using yFirstOsControl().

Returns:
- a pointer to a YOsControl object, corresponding to OS control currently online, or a null pointer if there are no more OS control to enumerate.
oscontrol→registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

`int registerValueCallback( UpdateCallback callback)`

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
Changes the logical name of the OS control.

```java
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.

```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
oscontrol → shutdown() oscontrol.shutdown()

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.
3.30. 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:

```
<script type='text/javascript' src='yocto_power.js'></script>
```

**Global functions**

`yFindPower(func)`
- Retrieves a electrical power sensor for a given identifier.

`yFirstPower()`
- Starts the enumeration of electrical power sensors currently accessible.

**YPower methods**

`power→calibrateFromPoints(rawValues, refValues)`
- Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

`power→describe()`
- Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

`power→get_advertisedValue()`
- Returns the current value of the electrical power sensor (no more than 6 characters).

`power→get_cosPhi()`
- Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

`power→get_currentRawValue()`
- Returns the uncalibrated, unrounded raw value returned by the sensor, 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 `MODULE_NAME.FUNCTION_NAME`.

`power→get_functionDescriptor()`
- Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
Returns the hardware identifier of the electrical power sensor, without reference to the module.

```
power->get_hardwareId()
```

Returns the unique hardware identifier of the electrical power sensor in the form SERIAL.FUNCTIONID.

```
power->get_highestValue()
```

Returns the maximal value observed for the electrical power since the device was started.

```
power->get_logFrequency()
```

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
power->get_logicalName()
```

Returns the logical name of the electrical power sensor.

```
power->get_lowestValue()
```

Returns the minimal value observed for the electrical power since the device was started.

```
power->get_meter()
```

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

```
power->get_meterTimer()
```

Returns the elapsed time since last energy counter reset, in seconds.

```
power->get_module()
```

Gets the YModule object for the device on which the function is located.

```
power->get_module_async(callback, context)
```

Gets the YModule object for the device on which the function is located (asynchronous version).

```
power->get_recordedData(startTime, endTime)
```

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
power->get_reportFrequency()
```

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
power->get_resolution()
```

Returns the resolution of the measured values.

```
power->get_unit()
```

Returns the measuring unit for the electrical power.

```
power->get_userData()
```

Returns the value of the userData attribute, as previously stored using method set_userData.

```
power->isOnline()
```

Checks if the electrical power sensor is currently reachable, without raising any error.

```
power->isOnline_async(callback, context)
```

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

```
power->load(msValidity)
```

Preloads the electrical power sensor cache with a specified validity duration.

```
power->loadCalibrationPoints(rawValues, refValues)
```

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
power->load_async(msValidity, callback, context)
```

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

```
power->nextPower()
```

Continues the enumeration of electrical power sensors started using yFirstPower().

```
power->registerTimedReportCallback(callback)
```

 Registers the callback function that is invoked on every periodic timed notification.

```
power->registerValueCallback(callback)
```

Registers the callback function that is invoked on every change of advertised value.

- **power.reset()**
  - Resets the energy counter.

- **power.set_highestValue(newval)**
  - Changes the recorded maximal value observed.

- **power.set_logFrequency(newval)**
  - Changes the datalogger recording frequency for this function.

- **power.set_logicalName(newval)**
  - Changes the logical name of the electrical power sensor.

- **power.set_lowestValue(newval)**
  - Changes the recorded minimal value observed.

- **power.set_reportFrequency(newval)**
  - Changes the timed value notification frequency for this function.

- **power.set_resolution(newval)**
  - Changes the resolution of the measured physical values.

- **power.set_userData(data)**
  - Stores a user context provided as argument in the userData attribute of the function.

- **power.wait_async(callback, context)**
  - Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YPower.FindPower()

YPower.FindPower()

Retrieves an 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 an 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.

```java
int calibrateFromPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
power.describe()

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form \( \text{TYPE} (\text{NAME}) = \text{SERIAL} . \text{FUNCTIONID} \).

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 electrical power sensor (ex: Relay(MyCustomName.relay1) = RELAYLO1-123456.relay1)
3. Reference

```
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`. 
Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

**double get_cosPhi()**

**Returns:**

A floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

On failure, throws an exception or returns `Y_COSPHI_INVALID`. 
Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

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_CURRENT_RAWVALUE_INVALID.
### 3. Reference

*YPower*

```java
power->get_currentValue()
power->currentValue() power.get_currentValue()
```

Returns the current value of the electrical power, in Watt, as a floating point number.

```java
double get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the electrical power, in Watt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
3. Reference

```java
YPower
power→getErrorMessage()
power→errorMessage()  
power.getErrorErrorMessage()
```

Returns the error message of the latest error with the electrical power sensor.

```java
String getErrorMessage()
```

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.

\textbf{Returns :} \\
a number corresponding to the code of the latest error that occurred while using the electrical power sensor object

\begin{verbatim}
int get_errorType()
\end{verbatim}
3. Reference

### YPower

#### power.get_friendlyName()

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

#### String get_friendlyName()

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for example: `MyCustomName.relay1`)

**Returns**:
- a string that uniquely identifies the electrical power sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.
 Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor() 

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
3. Reference

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.
3. Reference

**YPower**

```java
power→get_hardwareId()
power→hardwareId() power.get_hardwareId()
```

Returns the unique hardware identifier of the electrical power sensor 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 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`. 
3. Reference

\[
\begin{align*}
\text{YPower} & \rightarrow \text{get\_highestValue()} \\
\text{power} & \rightarrow \text{highestValue()} \\
\text{power.get\_highestValue()} &
\end{align*}
\]

Returns the maximal value observed for the electrical power since the device was started.

\begin{itemize}
\item \text{double get\_highestValue()} \\
\end{itemize}

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 \text{Y\_HIGHESTVALUE\_INVALID}.
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.
<table>
<thead>
<tr>
<th>Returns</th>
<th>String get_logicalName( )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a string corresponding to the logical name of the electrical power sensor.</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_LOGICALNAME_INVALID.</td>
</tr>
</tbody>
</table>
power\rightarrow get\_lowestValue() \hspace{1cm} YPower

power\rightarrow lowestValue() \hspace{1cm} power.get\_lowestValue()

Returns the minimal value observed for the electrical power since the device was started.

\textbf{double} \ \textbf{get\_lowestValue()} \hspace{1cm}

\textbf{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 \texttt{Y\_LOWESTVALUE\_INVALID}.
3. Reference

```c
YPower
power→get_meter()
power→meter() power.get_meter()
```

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

```c
double get_meter()
```

Note that this counter is reset at each start of the device.

**Returns**:

- a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns `Y_METER_INVALID`. 
Returns the elapsed time since last energy counter reset, in seconds.

```
int get_meterTimer()
```

**Returns:**
- an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns `Y_METERTIMER_INVALID`. 
power→get_module()  
get_module()  

YModule

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
power->get_recordedData()

power->recordedData()

power.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet get_recordedData( long startTime, long endTime)

The measures will be retrieved from the data logger, which must have been turned on at the desired
time. See the documentation of the DataSet class for information on how to get an overview of the
recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by
firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds
since January 1, 1970 UTC. The special value 0 can be used to include any 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.
YPower

power.get_reportFrequency()

power.reportFrequency()

calls YPower

power.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String get_reportFrequency()

Returns:

- a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.
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**.
3. Reference

```cpp
YPower

power.get_unit()
power.unit()power.get_unit()
```

Returns the measuring unit for the electrical power.

```cpp
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.
Checks if the electrical power sensor is currently reachable, without raising any error.

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

Returns:
- true if the electrical power sensor can be reached, and false otherwise
**power.load()**

Preloads the electrical power sensor cache with a specified validity duration.

```c
int load(long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

<table>
<thead>
<tr>
<th>Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>msValidity</strong></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><strong>YAPI_SUCCESS</strong></td>
<td>when the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
power->loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```c
int loadCalibrationPoints( ArrayList<Double> rawValues,
                          ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the enumeration of electrical power sensors started using `yFirstPower()`.

Returns:

- a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a null pointer if there are no more electrical power sensors to enumerate.
power.registerTimedReportCallback()

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

YPower

\textbf{power.registerValueCallback()}

Registers the callback function that is invoked on every change of advertised value.

\textbf{int registerValueCallback( UpdateCallback callback)}

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}{|l|}
\hline
\textbf{Parameters :} \\
\hline
\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}
3. Reference

`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.

**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.
3. Reference

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 electrical power sensor.

```java
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.

```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.
### 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.
3. Reference

**YPower**

`power->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
3.31. 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```
```
```

```pas```
```
```
```

```vb```
```
```
```

```cs```
```
```
```
```

```java```
import com.yoctopuce.YoctoAPI.YPressure;
```

```py```
```
```

### Global functions

**yFindPressure(func)**

Retrieves a pressure sensor for a given identifier.

**yFirstPressure()**

Starts the enumeration of pressure sensors currently accessible.

### YPressure methods

**pressure→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**pressure→describe()**

Returns a short text that describes unambiguously the instance of the pressure sensor in the form

```
TYPE(NAME)=SERIAL.FUNCTIONID
```

**pressure→get_advertisedValue()**

Returns the current value of the pressure sensor (no more than 6 characters).

**pressure→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

**pressure→get_currentValue()**

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

**pressure→getErrorMessage()**

Returns the error message of the latest error with the pressure sensor.

**pressure→getErrorType()**

Returns the numerical error code of the latest error with the pressure sensor.

**pressure→get_friendlyName()**

Returns a global identifier of the pressure sensor in the format

```
MODULE_NAME.FUNCTION_NAME
```

**pressure→get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

**pressure→get_functionId()**

Returns the hardware identifier of the pressure sensor, without reference to the module.

**pressure→get_hardwareId()**
Returns the unique hardware identifier of the pressure sensor in the form SERIAL.FUNCTIONID.

`pressure→get_highestValue()`
Returns the maximal value observed for the pressure since the device was started.

`pressure→get_logFrequency()`
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

`pressure→get_logicalName()`
Returns the logical name of the pressure sensor.

`pressure→get_lowestValue()`
Returns the minimal value observed for the pressure since the device was started.

`pressure→get_module()`
Gets the YModule object for the device on which the function is located.

`pressure→get_module_async(callback, context)`
Gets the YModule object for the device on which the function is located (asynchronous version).

`pressure→get_recordedData(startTime, endTime)`
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

`pressure→get_reportFrequency()`
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

`pressure→get_resolution()`
Returns the resolution of the measured values.

`pressure→get_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)`
### 3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pressure.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>pressure.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>pressure.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>pressure.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.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>
YPressure.FindPressure()

ypFindPressure() 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.isOnlinel() to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters:**

- `func` a string that uniquely characterizes the pressure sensor

**Returns:**

- a YPressure object allowing you to drive the pressure sensor.
YPressure.FirstPressure()

YPressure.FirstPressure()

Starts the enumeration of pressure sensors currently accessible.

Use the method YPressure.nextPressure() to iterate on next pressure sensors.

Returns:

- a pointer to a YPressure object, corresponding to the first pressure sensor currently online, or a null pointer if there are none.
pressure.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints( ArrayList<Double> rawValues,
                        ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
<table>
<thead>
<tr>
<th>pressure.describe()</th>
<th>YPressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a short text that describes unambiguously the instance of the pressure sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String describe()</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 &quot;unresolved&quot;, 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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns :</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string that describes the pressure sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)</td>
</tr>
</tbody>
</table>
pressure→get_advertisedValue()
purpose→advertisedValue()
purpose.get_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

String get_advertisedValue()

**Returns:**

- a string corresponding to the current value of the pressure sensor (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
YPressure

 pressure.get_currentRawValue()
 pressure.currentRawValue()
 pressure.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), 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 millibar (hPa), as a floating point number.

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`. 
### YPressure

**pressure\rightarrow get\_currentValue()**

**pressure\rightarrow currentValue()**

**pressure\. get\_currentValue()**

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

```java
double get_currentValue() {
    // Returns a floating point number corresponding to the current value of the pressure, in millibar (hPa), as a floating point number.
    // On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.
}
```
3. Reference

**YPressure**

```java
pressure.getError()  
pressure.getError()  
pressure.getError()  
```

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

pressure->get_errorType()
pressure->errorType()

pressure.get_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

int get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:

a number corresponding to the code of the latest error that occurred while using the pressure sensor object
YPressure

3. Reference

pressure→get_friendlyName()
p→friendlyName()
pressure.get_friendlyName()

Returns a global identifier of the pressure 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 pressure sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the pressure sensor (for example: `MyCustomName.relay1`)

**Returns:**
- a string that uniquely identifies the pressure sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
Pressure

pressure.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

- an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
pressure\rightarrow \text{get\_functionId()}
pressure\rightarrow \text{functionId()}
pressure.\text{get\_functionId()}

Returns the hardware identifier of the pressure sensor, without reference to the module.

String \text{get\_functionId()}()

For example \text{relay1}

Returns :
- a string that identifies the pressure sensor (ex: \text{relay1})

On failure, throws an exception or returns \text{Y\_FUNCTIONID\_INVALID}. 
pressure→get_hardwareId()
purpose→hardwareId()
purpose.get_hardwareId()

YPressure

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.
Returns the maximal value observed for the pressure since the device was started.

```
def get_highestValue(self) -> float:
    # Implementation details...
    return highest_value
```

**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.
### 3. Reference

```java
pressure → get_logicalName()
p=pressure→logicalName()
p=pressure.get_logicalName()
```

Returns the logical name of the pressure sensor.

**String get_logicalName()**

**Returns:**
- a string corresponding to the logical name of the pressure sensor.
- On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
3. Reference

**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.
YPressure

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.
3. Reference

**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`. 


```cpp
pressure->get_resolution()
presure->resolution()
presure.get_resolution()
```

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`. 
**YPressure**

```java
pressure.get_unit()
prefix.unit() pressure.get_unit()
```

Returns the measuring unit for the pressure.

**String get_unit()**

**Returns:**
- a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns `Y_UNIT_INVALID`.
pressure→get_userData()
pressure→userData() pressure.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object `get_userData()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns:**
- the object stored previously by the caller.
pressure→isOnline() pressure.isOnline()  

Checks if the pressure sensor is currently reachable, without raising any error.

boolean isOnline( )

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

**Returns :**
- true if the pressure sensor can be reached, and false otherwise
Preloads the pressure sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
pressure->loadCalibrationPoints() 

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
<table>
<thead>
<tr>
<th>pressure.nextPressure()</th>
<th>YPressure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Continues the enumeration of pressure sensors started using yFirstPressure().</strong></td>
<td></td>
</tr>
<tr>
<td><strong>YPressure nextPressure()</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Returns:</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
registerTimedReportCallback()  
pressure.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

**pressure→registerValueCallback()**

 Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>callback</code> the callback 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.</td>
</tr>
</tbody>
</table>
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.
pressure.set_logicalName()

Changes the logical name of the pressure 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 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.

**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.
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. 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;
require_once('yocto_pwminput.php');
```
```cpp
#include "yocto_pwminput.h"
```
```pas
uses yocto_pwminput;
```
```vb
yocto_pwminput.vb
```
```cs
yocto_pwminput.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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwminput.get_functionId()</code></td>
<td>Returns the hardware identifier of the voltage sensor, without reference to the module.</td>
</tr>
<tr>
<td><code>pwminput.get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>pwminput.get_highestValue()</code></td>
<td>Returns the maximal value observed for the voltage since the device was started.</td>
</tr>
<tr>
<td><code>pwminput.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>pwminput.get_logicalName()</code></td>
<td>Returns the logical name of the voltage sensor.</td>
</tr>
<tr>
<td><code>pwminput.get_lowestValue()</code></td>
<td>Returns the minimal value observed for the voltage since the device was started.</td>
</tr>
<tr>
<td><code>pwminput.get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>pwminput.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>pwminput.get_period()</code></td>
<td>Returns the PWM period in milliseconds.</td>
</tr>
<tr>
<td><code>pwminput.get_pulseCounter()</code></td>
<td>Returns the pulse counter value.</td>
</tr>
<tr>
<td><code>pwminput.get_pulseDuration()</code></td>
<td>Returns the PWM pulse length in milliseconds, as a floating point number.</td>
</tr>
<tr>
<td><code>pwminput.get_pulseTimer()</code></td>
<td>Returns the timer of the pulses counter (ms)</td>
</tr>
<tr>
<td><code>pwminput.get_pwmReportMode()</code></td>
<td>Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.</td>
</tr>
<tr>
<td><code>pwminput.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>pwminput.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>pwminput.get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>pwminput.get_unit()</code></td>
<td>Returns the measuring unit for the values returned by get_currentValue and callbacks.</td>
</tr>
<tr>
<td><code>pwminput.get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><code>pwminput.isOnline()</code></td>
<td>Checks if the voltage sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>pwminput.isOnline_async(callback, context)</code></td>
<td>Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>pwminput.load(msValidity)</code></td>
<td>Preloads the voltage sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>pwminput.loadCalibrationPoints(rawValues, refValues)</code></td>
<td></td>
</tr>
</tbody>
</table>
3. Reference

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```c
pwminput->load_async(msValidity, callback, context)
```

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

```c
pwminput->nextPwmInput()
```

Continues the enumeration of voltage sensors started using `yFirstPwmInput()`.

```c
pwminput->registerTimedReportCallback(callback)
```

Registers the callback function that is invoked on every periodic timed notification.

```c
pwminput->registerValueCallback(callback)
```

Registers the callback function that is invoked on every change of advertised value.

```c
pwminput->resetCounter()
```

Returns the pulse counter value as well as his timer.

```c
pwminput->set_highestValue(newval)
```

Changes the recorded maximal value observed.

```c
pwminput->set_logFrequency(newval)
```

Changes the datalogger recording frequency for this function.

```c
pwminput->set_logicalName(newval)
```

Changes the logical name of the voltage sensor.

```c
pwminput->set_lowestValue(newval)
```

Changes the recorded minimal value observed.

```c
pwminput->set_pwmReportMode(newval)
```

Modify the parameter type (frequency/duty cycle, pulse width ou edge count) returned by the `get_currentValue` function and callbacks.

```c
pwminput->set_reportFrequency(newval)
```

Changes the timed value notification frequency for this function.

```c
pwminput->set_resolution(newval)
```

Changes the resolution of the measured physical values.

```c
pwminput->set_userData(data)
```

Stores a user context provided as argument in the `userData` attribute of the function.

```c
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.
**YPwmInput.FirstPwmInput()**

**yFirstPwmInput()**

*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.
3. Reference

**pwminput → calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
pwminput.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 it the name used for the first access to the 
function, SERIAL is the serial number of the module if the module is connected or "unresolved", 
and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, 
this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the 
module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module 
has not yet been connected. This method does not trigger any USB or TCP transaction and can 
therefore be used in a debugger.

Returns:

a string that describes the voltage sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1- 
123456.relay1)
3. Reference

\[ \text{YPwmInput} \]
\[ \text{pwminput} \rightarrow \text{get\_advertisedValue()} \]
\[ \text{pwminput} \rightarrow \text{advertisedValue()} \]
\[ \text{pwminput.get\_advertisedValue()} \]

Returns the current value of the voltage sensor (no more than 6 characters).

\textbf{String get\_advertisedValue()}

\textbf{Returns :}

- a string corresponding to the current value of the voltage sensor (no more than 6 characters).

On failure, throws an exception or returns \texttt{Y\_ADVERTISEDVALUE\_INVALID}.
ypminput→get_currentRawValue()
ypminput→currentRawValue()
ypminput.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

`pwminput→get_currentValue()`  
`pwminput→currentValue()`  
`pwminput.get_currentValue()`  

Returns the current value of PwmInput feature as a floating point number.

```java
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`. 
Returns the PWM duty cycle, in per cents.

double get_dutyCycle()

**Returns:**
- a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns **Y_DUTYCYCLE_INVALID**.
3. Reference

```java
YPwmInput
pwminput→get_errorMessage()
pwminput→errorMessage()
pwminput.get_errorMessage()
```

Returns the error message of the latest error with the voltage sensor.

```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 voltage sensor object
`pwminput->get_errorType()`  
`pwminput->errorType()`  
`pwminput.get_errorType()`

Returns the numerical error code of the latest error with the voltage sensor.

```
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns**:

A number corresponding to the code of the latest error that occurred while using the voltage sensor object.
3. Reference

**YPwmInput**

```c
pwminput->get_frequency()
pwminput->frequency()
pwminput.get_frequency()
```

Returns the PWM frequency in Hz.

```c
double get_frequency() {
    // Function implementation
}
```

**Returns:**

a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`. 
pwminput→get_friendlyName()
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.
ypminput→get_functionDescriptor()  YPwmInput
ypminput→functionDescriptor()
ypminput.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor()  

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
3. Reference

YPwmInput

pwminput→get_functionId()

pwminput→functionId()

pwminput.get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

String get_functionId()

For example relay1

Returns:

a string that identifies the voltage sensor (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
3. Reference

**YPwmInput**

```java
pwminput→get_hardwareId()
pwminput→hardwareId()
pwinput_.get_hardwareId()
```

Returns the unique hardware identifier of the voltage sensor in the form **SERIAL.FUNCTIONID**.

**String get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor (for example **RELAYLO1-123456.relay1**).

**Returns :**

- a string that uniquely identifies the voltage sensor (ex: **RELAYLO1-123456.relay1**)

On failure, throws an exception or returns **Y_HARDWAREID_INVALID**.
Returns the maximal value observed for the voltage since the device was started.

**double get_highestValue()**

**Returns:**
- a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`. 
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 voltage sensor.

**String get_logicalName()**

- **Returns:**
  - a string corresponding to the logical name of the voltage sensor.

- On failure, throws an exception or returns **Y_LOGICALNAME_INVALID**.
3. Reference

YPwmInput

pwminput.get_lowestValue()

pwminput→lowestValue()
pwminput.get_lowestValue() ()

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.
YPwmInput

pwminput→get_module()
pwminput→module()pwminput.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
`pwminput.get_period()`

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 `YPWMLib.Y_PERIOD_INVALID`. 
pwminput→get_pulseCounter()
pwminput→pulseCounter()
pwminput.get_pulseCounter()

Returns the pulse counter value.

`long get_pulseCounter()`

Actually that counter is incremented twice per period. That counter is limited to 1 billions

**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.

```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.
```
### YPwmInput

**pwminput.get pulseTimer()**

Returns the timer of the pulses counter (ms)

```cpp
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.
**3. Reference**

`ypwminput->get_pwmReportMode()`

`ypwminput->pwmReportMode()`

`ypwminput.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_PULSEDEUATION` 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`. 
**YPwmInput**

**pwminput->get_recordedData()**  
**pwminput->recordedData()**  
**pwminput.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><strong>endTime</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>an instance of <a href="#">YDataSet</a>, providing access to historical data. Past measures can be loaded progressively using methods from the <a href="#">YDataSet</a> object.</td>
</tr>
</tbody>
</table>
ypminput→get_reportFrequency()
pwminput→reportFrequency()
pwminput.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.
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

YPwmInput

pwminput → get_unit()
pwminput → unit() pwminput.get_unit()

Returns the measuring unit for the values returned by get_currentValue and callbacks.

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`.

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.

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

**ypminput.load()**

Preloads the voltage sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- **msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
**3. Reference**

**YPwmInput**

**pwminput.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the enumeration of voltage sensors started using `ypwmInput.firstPwmInput()`.

**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.
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.
pwminput.registerValueCallback()  

Registers the callback function that is invoked on every change of advertised value.

```cpp
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
### Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwinput.resetCounter()</code></td>
<td>Returns the pulse counter value as well as his timer</td>
</tr>
</tbody>
</table>

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
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 voltage sensor.

```java
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.

```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.
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 `YPWMREPORTMODE_PWM_DUTYCYCLE`, `YPWMREPORTMODE_PWM_FREQUENCY`, `YPWMREPORTMODE_PWM_PULSEDETECTION` and `YPWMREPORTMODE_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.

**Parameters:**

- `newval` a floating point number corresponding to the resolution of the measured physical values

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
pwminput→set_userData()  
 pwminput→setUserData()  
 pwminput.set_userData()  

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
3.33. 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;
require_once('yocto_pwmoutput.php');
#include "yocto_pwmoutput.h"
#import "yocto_pwmoutput.h"
uses yocto_pwmoutput;
yocto_pwmoutput.vb
yocto_pwmoutput.cs
```

```php
require_once('yocto_pwmoutput.php');
```

```cpp
#include "yocto_pwmoutput.h"
```

```m
uses yocto_pwmoutput;
```

```pas
uses yocto_pwmoutput;
```

```vb
yocto_pwmoutput.vb
```

```java
import com.yoctopuce.YoctoAPI.YPwmOutput;
```

```py
from yocto_pwmoutput import *
```

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindPwmOutput</strong></td>
<td>(func) Retrieves a PWM for a given identifier.</td>
</tr>
<tr>
<td><strong>yFirstPwmOutput</strong></td>
<td>Starts the enumeration of PWMs currently accessible.</td>
</tr>
</tbody>
</table>

### YPwmOutput methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pwmoutput.describe</strong></td>
<td>Returns a short text that describes unambiguously the instance of the PWM in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><strong>pwmoutput.dutyCycleMove</strong></td>
<td>(target, ms_duration) Performs a smooth change of the pulse duration toward a given value.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_advertisedValue</strong></td>
<td>Returns the current value of the PWM (no more than 6 characters).</td>
</tr>
<tr>
<td><strong>pwmoutput.get_dutyCycle</strong></td>
<td>Returns the PWM duty cycle, in per cents.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_dutyCycleAtPowerOn</strong></td>
<td>Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100</td>
</tr>
<tr>
<td><strong>pwmoutput.get_enabled</strong></td>
<td>Returns the state of the PWMs.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_enabledAtPowerOn</strong></td>
<td>Returns the state of the PWM at device power on.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_errorMessage</strong></td>
<td>Returns the error message of the latest error with the PWM.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_errorType</strong></td>
<td>Returns the numerical error code of the latest error with the PWM.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_frequency</strong></td>
<td>Returns the PWM frequency in Hz.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_friendlyName</strong></td>
<td>Returns a global identifier of the PWM in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><strong>pwmoutput.get_functionDescriptor</strong></td>
<td></td>
</tr>
</tbody>
</table>
3. Reference

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```csharp
pwmoutput→get_functionId()
```

- Returns the hardware identifier of the PWM, without reference to the module.

```csharp
pwmoutput→get_hardwareId()
```

- Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

```csharp
pwmoutput→get_logicalName()
```

- Returns the logical name of the PWM.

```csharp
pwmoutput→get_module()
```

- Gets the YModule object for the device on which the function is located.

```csharp
pwmoutput→get_module_async(callback, context)
```

- Gets the YModule object for the device on which the function is located (asynchronous version).

```csharp
pwmoutput→get_period()
```

- Returns the PWM period in milliseconds.

```csharp
pwmoutput→get_pulseDuration()
```

- Returns the PWM pulse length in milliseconds, as a floating point number.

```csharp
pwmoutput→get_userData()
```

- Returns the value of the userData attribute, as previously stored using method set_userData.

```csharp
pwmoutput→isOnline()
```

- Checks if the PWM is currently reachable, without raising any error.

```csharp
pwmoutput→isOnline_async(callback, context)
```

- Checks if the PWM is currently reachable, without raising any error (asynchronous version).

```csharp
pwmoutput→load(msValidity)
```

- Preloads the PWM cache with a specified validity duration.

```csharp
pwmoutput→load_async(msValidity, callback, context)
```

- Preloads the PWM cache with a specified validity duration (asynchronous version).

```csharp
pwmoutput→nextPwmOutput()
```

- Continues the enumeration of PWMs started using yFirstPwmOutput().

```csharp
pwmoutput→pulseDurationMove(ms_target, ms_duration)
```

- Performs a smooth transition of the pulse duration toward a given value.

```csharp
pwmoutput→registerValueCallback(callback)
```

- Registers the callback function that is invoked on every change of advertised value.

```csharp
pwmoutput→set_dutyCycle(newval)
```

- Changes the PWM duty cycle, in per cents.

```csharp
pwmoutput→set_dutyCycleAtPowerOn(newval)
```

- Changes the PWM duty cycle at device power on.

```csharp
pwmoutput→set_enabled(newval)
```

- Stops or starts the PWM.

```csharp
pwmoutput→set_enabledAtPowerOn(newval)
```

- Changes the state of the PWM at device power on.

```csharp
pwmoutput→set_frequency(newval)
```

- Changes the PWM frequency.

```csharp
pwmoutput→set_logicalName(newval)
```

- Changes the logical name of the PWM.

```csharp
pwmoutput→set_period(newval)
```

- Changes the PWM period in milliseconds.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwmoutput.set_pulseDuration</code></td>
<td>Changes the PWM pulse length, in milliseconds.</td>
</tr>
<tr>
<td><code>pwmoutput.set_userData</code></td>
<td>Stores a user context provided as argument in the <code>userData</code> attribute.</td>
</tr>
<tr>
<td><code>pwmoutput.wait_async</code></td>
<td>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided <code>callback</code> function.</td>
</tr>
</tbody>
</table>
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()

Starts the enumeration of PWMs currently accessible.

Use the method \texttt{YPwmOutput.nextPwmOutput()} to iterate on next PWMs.

\textbf{Returns:}

\begin{itemize}
\item a pointer to a \texttt{YPwmOutput} object, corresponding to the first PWM currently online, or a \texttt{null} pointer if there are none.
\end{itemize}
 PWM

### pwmoutput.describe()

Returns a short text that describes unambiguously the instance of the PWM 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

- \( \text{Relay(MyCustomName.relay1)} = \text{RELAYLO1-123456.relay1} \) if the module is already connected
- \( \text{Relay(BadCustomName.relay1)} = \text{unresolved} \) 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 PWM (ex: \( \text{Relay(MyCustomName.relay1)} = \text{RELAYLO1-123456.relay1} \))
YPwmOutput

Perform a smooth change of the pulse duration toward a given value.

```c
int dutyCycleMove( double target, int ms_duration)
```

**Parameters:**
- `target` new duty cycle at the end of the transition (floating-point number, between 0 and 1)
- `ms_duration` total duration of the transition, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.
- On failure, throws an exception or returns a negative error code.
### Reference

**YPwmOutput**

- `pwmoutput→get_advertisedValue()`
- `pwmoutput→advertisedValue()`
- `pwmoutput.get_advertisedValue()`

Returns the current value of the PWM (no more than 6 characters).

#### String `get_advertisedValue()`

**Returns:**
- a string corresponding to the current value of the PWM (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
Returns the PWM duty cycle, in per cents.

double get_dutyCycle()

Returns:
- a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns Y_DUTYCYCLE_INVALID.
Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

```cpp
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**

- `get_enabledAtPowerOn()`
- `enabledAtPowerOn()`
- `pwmoutput.get_enabledAtPowerOn()`

Returns the state of the PWM at device power on.

```c
int get_enabledAtPowerOn()
```

**Returns:**

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`. 
Returns the error message of the latest error with the PWM.

String `get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

a string corresponding to the latest error message that occurred while using the PWM object
3. Reference

`ypwmoutput.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.
### YPwmOutput

**pwmoutput→get_frequency()**

**pwmoutput→frequency()**

**pwmoutput.get_frequency()**

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**.
Returns a global identifier of the PWM in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName( )

The returned string uses the logical names of the module and of the PWM if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM (for 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.
### YPwmOutput

#### YPwmOutput

`get_functionDescriptor()`

`functionDescriptor()`

`get_functionDescriptor()`

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**String get_functionDescriptor()**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>an identifier of type <code>YFUN_DESCR</code>.</td>
</tr>
</tbody>
</table>

If the function has never been contacted, the returned value is `Y_FUNCTION_DESCRIPTOR_INVALID`. 
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.

**String get_hardwareId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM (for example RELAYLO1-123456.relay1).

**Returns :**

a string that uniquely identifies the PWM (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
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`. 
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
Returns the PWM period in milliseconds.

```cpp
double get_period() // YPwmOutput

Returns:
  a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns Y_PERIOD_INVALID.
**Reference**

`pwmoutput.get_pulseDuration()`  
Returns the PWM pulse length in milliseconds, as a floating point number.

```c
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

YPwmOutput

pwmoutput→get_userData()
pwmoutput→userData()
pwmoutput.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.
Checks if the PWM is currently reachable, without raising any error.

**Boolean isOnline()**

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

**Returns:**
- `true` if the PWM can be reached, and `false` otherwise
Preloads the PWM cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

**Returns**: a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.
3. Reference

YPwmOutput

pwmoutput.pulseDurationMove()

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.
ypwoutput.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- **callback** the callback function to call, or a null pointer. The callback function should take two arguments:
  - the function object of which the value has changed, and
  - the character string describing the new advertised value.
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.

`int set_dutyCycleAtPowerOn( double newval)`

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters:**
- `newval` a floating point number corresponding to the PWM duty cycle at device power on

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
pwmoutput\rightarrow\text{set\_enabled()}
pwmoutput\rightarrow\text{setEnabled()}
pwmoutput.set\_enabled()

Stops or starts the PWM.

\texttt{int set\_enabled( int newval)}

\begin{itemize}
  \item **Parameters:**
    \begin{itemize}
      \item \texttt{newval} either \texttt{Y\_ENABLED\_FALSE} or \texttt{Y\_ENABLED\_TRUE}
    \end{itemize}
  \end{itemize}

\begin{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.
Changes the state of the PWM at device power on.

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.

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.

```c
int set_period( double newval)
```

**Parameters:**
- `newval` a floating point number corresponding to the PWM period in milliseconds

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the PWM pulse length, in milliseconds.

```c
int set_pulseDuration( double newval)
```

A pulse length cannot be longer than period, otherwise it is truncated.

**Parameters :**
- `newval` a floating point number corresponding to the PWM pulse length, in milliseconds

**Returns :**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
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
3.34. 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;
require_once('yocto_pwmpowersource.php');
#include "yocto_pwmpowersource.h"
#import "yocto_pwmpowersource.h"
uses yocto_pwmpowersource;
yocto_pwmpowersource.vb
yocto_pwmpowersource.cs
yocto_pwmpowersource.php
import com.yoctopuce.YoctoAPI.YPwmPowerSource;
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).
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_powerMode()</code></td>
<td>Returns the selected power source for the PWM on the same device.</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 voltage source is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>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>load(msValidity)</code></td>
<td>Preloads the voltage source cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>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>nextPwmPowerSource()</code></td>
<td>Continues the enumeration of Voltage sources started using <code>yFirstPwmPowerSource()</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 voltage source.</td>
</tr>
<tr>
<td><code>set_powerMode(newval)</code></td>
<td>Changes the PWM power source.</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>
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.
ypmpowersource.describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form

\[ \text{TYPE}(\text{NAME})=\text{SERIAL.FUNCTIONID} \]

String \text{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.

\text{Returns :}

a string that describes the voltage source (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
Returns the current value of the voltage source (no more than 6 characters).

String `get_advertisedValue()`

**Returns:**
- a string corresponding to the current value of the voltage source (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
Returns the error message of the latest error with the voltage source.

**String get(errorMessage())**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

a string corresponding to the latest error message that occurred while using the voltage source object
Returns the numerical error code of the latest error with the voltage source.

\[
\text{int get\_errorType( )}
\]

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**
- a number corresponding to the code of the latest error that occurred while using the voltage source object
Returns a global identifier of the voltage source in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName()  

The returned string uses the logical names of the module and of the voltage source if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage source (for example: MyCustomName.relay1)

Returns :

a string that uniquely identifies the voltage source using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
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

**YPwmPowerSource**

**pwmpowersource→get_functionId()**  
**pwmpowersource→functionId()**  
**pwmpowersource.get_functionId()**

Returns the hardware identifier of the voltage source, without reference to the module.

```java
String get_functionId()
```

For example *relay1*

<table>
<thead>
<tr>
<th><strong>Returns:</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a string that identifies the voltage source (ex: <em>relay1</em>)</td>
<td></td>
</tr>
<tr>
<td>On failure, throws an exception or returns <strong>Y_FUNCTIONID_INVALID</strong>.</td>
<td></td>
</tr>
</tbody>
</table>
The method `get_hardwareId()` 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`. 
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.
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
**YPwmPowerSource**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwmpowersource.get_powerMode()</td>
<td>Returns the selected power source for the PWM on the same device</td>
</tr>
<tr>
<td>pwmpowersource.powerMode()</td>
<td></td>
</tr>
<tr>
<td>pwmpowersource.get_powerMode()</td>
<td></td>
</tr>
</tbody>
</table>

**Returns :**

A value among Y_POWERMODE_USB_5V, Y_POWERMODE_USB_3V, Y_POWERMODE_EXT_V and Y_POWERMODE_OPNDRN corresponding to the selected power source for the PWM on the same device.

On failure, throws an exception or returns Y_POWERMODE_INVALID.
3. Reference

`pwmpowersource→getUserData()`

`pwmpowersource→userData()`

`pwmpowersource.getUserData()`

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.
**YPwmPowerSource**

`pwmpowersource.isOnline()`  

Checks if the voltage source is currently reachable, without raising any error.

```java
boolean isOnline()
```

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

**Returns:**

- `true` if the voltage source can be reached, and `false` otherwise.
Preloads the voltage source cache with a specified validity duration.

```c
int load(long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- **msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the enumeration of Voltage sources started using `ypFirstPwmPowerSource()`.

**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( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
Changes the logical name of the voltage source.

```java
int set_logicalName(String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters:**
- `newval` a string corresponding to the logical name of the voltage source.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
Changes the PWM power source.

### 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.
**3. Reference**

```cpp
pwmpowersource->set_userData()  
pwmpowersource->setUserData()  
pwmpowersource.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
3.35. 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
#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 *
```

<table>
<thead>
<tr>
<th>Global functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yFindQt</strong>&lt;br&gt;(func)</td>
</tr>
<tr>
<td>Retrieves a quaternion component for a given identifier.</td>
</tr>
<tr>
<td><strong>yFirstQt</strong>()</td>
</tr>
<tr>
<td>Starts the enumeration of quaternion components currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YQt methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>qt→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>qt→describe</strong>()</td>
</tr>
<tr>
<td>Returns a short text that describes unambiguously the instance of the quaternion component in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><strong>qt→get_advertisedValue</strong>()</td>
</tr>
<tr>
<td>Returns the current value of the quaternion component (no more than 6 characters).</td>
</tr>
<tr>
<td><strong>qt→get_currentRawValue</strong>()</td>
</tr>
<tr>
<td>Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.</td>
</tr>
<tr>
<td><strong>qt→get_currentValue</strong>()</td>
</tr>
<tr>
<td>Returns the current value of the value, in units, as a floating point number.</td>
</tr>
<tr>
<td><strong>qt→get_errorMessage</strong>()</td>
</tr>
<tr>
<td>Returns the error message of the latest error with the quaternion component.</td>
</tr>
<tr>
<td><strong>qt→get_errorType</strong>()</td>
</tr>
<tr>
<td>Returns the numerical error code of the latest error with the quaternion component.</td>
</tr>
<tr>
<td><strong>qt→get_friendlyName</strong>()</td>
</tr>
<tr>
<td>Returns a global identifier of the quaternion component in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><strong>qt→get_functionDescriptor</strong>()</td>
</tr>
<tr>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><strong>qt→get_functionId</strong>()</td>
</tr>
<tr>
<td>Returns the hardware identifier of the quaternion component, without reference to the module.</td>
</tr>
<tr>
<td><strong>qt→get_hardwareId</strong>()</td>
</tr>
</tbody>
</table>
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qt→get_highestValue()</code></td>
<td>Returns the maximal value observed for the value since the device was started.</td>
</tr>
<tr>
<td><code>qt→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>qt→get_logicalName()</code></td>
<td>Returns the logical name of the quaternion component.</td>
</tr>
<tr>
<td><code>qt→get_lowestValue()</code></td>
<td>Returns the minimal value observed for the value since the device was started.</td>
</tr>
<tr>
<td><code>qt→get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>qt→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>qt→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>qt→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>qt→get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>qt→get_unit()</code></td>
<td>Returns the measuring unit for the value.</td>
</tr>
<tr>
<td><code>qt→get_userData()</code></td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><code>qt→isOnline()</code></td>
<td>Checks if the quaternion component is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>qt→isOnline_async(callback, context)</code></td>
<td>Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>qt→load(msValidity)</code></td>
<td>Preloads the quaternion component cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>qt→loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method calibrateFromPoints.</td>
</tr>
<tr>
<td><code>qt→load_async(msValidity, callback, context)</code></td>
<td>Preloads the quaternion component cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>qt→nextQt()</code></td>
<td>Continues the enumeration of quaternion components started using yFirstQt().</td>
</tr>
<tr>
<td><code>qt→registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>qt→registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>qt→set_highestValue(newval)</code></td>
<td>Changes the recorded maximal value observed.</td>
</tr>
<tr>
<td><code>qt→set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>qt→set_logicalName(newval)</code></td>
<td></td>
</tr>
</tbody>
</table>
Changes the logical name of the quaternion component.

\[ \text{qt} \rightarrow \text{set_lowestValue(newval)} \]
- Changes the recorded minimal value observed.

\[ \text{qt} \rightarrow \text{set_reportFrequency(newval)} \]
- Changes the timed value notification frequency for this function.

\[ \text{qt} \rightarrow \text{set_resolution(newval)} \]
- Changes the resolution of the measured physical values.

\[ \text{qt} \rightarrow \text{set_userData(data)} \]
- Stores a user context provided as argument in the userData attribute of the function.

\[ \text{qt} \rightarrow \text{wait_async(callback, context)} \]
- Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YQt.FindQt()

YQt.FindQt(String func)

Retrieves a quaternion component 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 quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method YQt.isOnline() to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters:**
- **func** a string that uniquely characterizes the quaternion component

**Returns:**
- a YQt object allowing you to drive the quaternion component.
YQt.FirstQt()

**YQt.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.
Yqt → calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
qt.describe() Qt

Returns a short text that describes unambiguously the instance of the quaternion component 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} its 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 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.

\text{Returns : a string that describes the quaternion component (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
YQt

qt→get_advertisedValue()
qt→advertisedValue() qt.get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

String get_advertisedValue() 

Returns:

a string corresponding to the current value of the quaternion component (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
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.

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`. 
3. Reference

qt→get_errorMessage()
qt→errorMessage()

YQt

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.
get_errorType() Returns the numerical error code of the latest error with the quaternion component.

int get_errorType() Returns: a number corresponding to the code of the latest error that occurred while using the quaternion component object.

This method is mostly useful when using the Yoctopuce library with exceptions disabled.
YQt qt->get_friendlyName()
qt->friendlyName() qt.get_friendlyName()

Returns a global identifier of the quaternion component in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName()

The returned string uses the logical names of the module and of the quaternion component if they are defined, otherwise the serial number of the module and the hardware identifier of the quaternion component (for example: MyCustomName.relay1)

Returns:
- A string that uniquely identifies the quaternion component using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
qt \rightarrow \text{get\_functionDescriptor()}
qt \rightarrow \text{functionDescriptor()}
qt.\text{get\_functionDescriptor()}

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

\textbf{String get\_functionDescriptor() }

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

\textbf{Returns :}
\begin{itemize}
  \item an identifier of type YFUN_DESCR.
\end{itemize}

\textbf{If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.}
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.

String `get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quaternion component (for example `RELAYLO1-123456.relay1`).

**Returns:**

- a string that uniquely identifies the quaternion component (ex: `RELAYLO1-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
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()
qt->logFrequency()qt.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String get_logFrequency( )

Returns:
a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.
Returns the logical name of the quaternion component.

String \texttt{get\_logicalName()} 

\begin{itemize}
  \item Returns: a string corresponding to the logical name of the quaternion component.
  \item On failure, throws an exception or returns \texttt{Y\_LOGICALNAME\_INVALID}.
\end{itemize}
YQt
qt→getLowestValue()
qt→lowestValue()qt.getLowestValue()

Returns the minimal value observed for the value since the device was started.

double getLowestValue()

**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.
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
qt\rightarrow get\_recordedData()\quad YQt\quad qt\rightarrow 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.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startTime</strong> 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> 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>

<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.

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`. 
YQt

### qt\_get\_resolution()

```
qt->resolution()qt.get_resolution()
```

Returns the resolution of the measured values.

```python
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 value.

**String get_unit()**

**Returns:**
- a string corresponding to the measuring unit for the value

On failure, throws an exception or returns Y_UNIT_INVALID.
qt->get_userData()

qt->userData() \texttt{qt UserData()}

Returns the value of the userData attribute, as previously stored using method \texttt{set_userData}.

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}:

\begin{itemize}
  \item the object stored previously by the caller.
\end{itemize}
YQt

Checks if the quaternion component is currently reachable, without raising any error.

```java
boolean isOnline()
```

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

**Returns:**

true if the quaternion component can be reached, and false otherwise
qt → load()  
YQt

Preloads the quaternion component cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
qt->loadCalibrationPoints()
qt.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```cpp
int loadCalibrationPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
qt\rightarrow\text{nextQt}()\quad YQt

Continues the enumeration of quaternion components started using \text{yFirstQt}().

\begin{tabular}{|p{0.9\textwidth}|}
\hline
\textbf{YQt nextQt( )} \\
\hline
\textbf{Returns :} \\
\hspace{1em} a pointer to a \textit{YQt} object, corresponding to a quaternion component currently online, or a \textit{null} pointer if there are no more quaternion components to enumerate. \\
\hline
\end{tabular}
qt->registerTimedReportCallback() YQt
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.
qt\rightarrow\text{registerValueCallback()}
qt.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

\begin{verbatim}
int registerValueCallback( UpdateCallback 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.

**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.
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.
YQt

\begin{quote}
qt\texttt{\_set\_log\_Frequency()}
\end{quote}

\begin{quote}
qt\texttt{\_set\_Log\_Frequency()}
\end{quote}

\begin{quote}
qt.set\_log\_Frequency()
\end{quote}

Changes the datalogger recording frequency for this function.

\begin{quote}
\textbf{int set\_log\_Frequency( String newval)}
\end{quote}

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".

\begin{quote}
\begin{tabular}{|l|}
\hline
\textbf{Parameters} : \\
\texttt{newval} a string corresponding to the datalogger recording frequency for this function \\
\hline
\textbf{Returns} : \\
\texttt{YAPI\_SUCCESS} if the call succeeds. \\
\hline
\end{tabular}
\end{quote}

On failure, throws an exception or returns a negative error code.
Changes the logical name of the quaternion component.

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters:**
- `newval`: a string corresponding to the logical name of the quaternion component.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
qt->set_lowestValue()
qt->setLowestValue() \texttt{qt.set\_lowestValue()}

Changes the recorded minimal value observed.

\begin{verbatim}
int \texttt{set\_lowestValue}( double \texttt{newval})
\end{verbatim}

**Parameters:**
- \texttt{newval} a floating point number corresponding to the recorded minimal value observed

**Returns:**
- \texttt{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.

```java
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

`qt->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
3.36. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even across power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

```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"
```
```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()**
3. Reference

- **YModule** object for the device on which the function is located.

**get_module_async(callback, context)**
- Gets the **YModule** object for the device on which the function is located (asynchronous version).

**get_timeSet()**
- Returns true if the clock has been set, and false otherwise.

**get_unixTime()**
- Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

**get_userData()**
- Returns the value of the userData attribute, as previously stored using method `set_userData`.

**get_utcOffset()**
- Returns the number of seconds between current time and UTC time (time zone).

**isOnline()**
- Checks if the clock is currently reachable, without raising any error.

**isOnline_async(callback, context)**
- Checks if the clock is currently reachable, without raising any error (asynchronous version).

**load(msValidity)**
- Preloads the clock cache with a specified validity duration.

**load_async(msValidity, callback, context)**
- Preloads the clock cache with a specified validity duration (asynchronous version).

**nextRealTimeClock()**
- Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

**registerValueCallback(callback)**
- Registers the callback function that is invoked on every change of advertised value.

**set_logicalName(newval)**
- Changes the logical name of the clock.

**set_unixTime(newval)**
- Changes the current time.

**set_userData(data)**
- Stores a user context provided as argument in the userData attribute of the function.

**set_utcOffset(newval)**
- Changes the number of seconds between current time and UTC time (time zone).

**wait_async(callback, context)**
- Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YRealTimeClock.FindRealTimeClock()
YRealTimeClock.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.

YRealTimeClock FirstRealTimeClock()

Use the method YRealTimeClock.nextRealTimeClock() to iterate on next clocks.

**Returns:**

- a pointer to a YRealTimeClock object, corresponding to the first clock currently online, or a null pointer if there are none.
3. Reference

YRealTimeClock

realtimeclock.describe()

realtimeclock.describe()

Returns a short text that describes unambiguously the instance of the clock in the form

TYPE(NAME)=SERIAL.FUNCTIONID.

String describe()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:

a string that describes the clock (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
3. Reference

realtimeclock→get_advertisedValue()  YRealTimeClock
realtimeclock→advertisedValue()  
realtimeclock.get_advertisedValue()  

Returns the current value of the clock (no more than 6 characters).

String get_advertisedValue()

**Returns**:

- a string corresponding to the current value of the clock (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
3. Reference

YRealTimeClock

- `realtimeclock→get_dateTime()`
- `realtimeclock→dateTime()`
- `realtimeclock.get_dateTime()`

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

**String get_dateTime( )**

**Returns:**

A string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns `Y_DATETIME_INVALID`. 
3. Reference

`realtimeclock → get_errorMessage()`  
`realtimeclock → errorMessage()`  
`realtimeclock.get_errorMessage()`  

Returns the error message of the latest error with the clock.

**String `get_errorMessage()`**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

- a string corresponding to the latest error message that occurred while using the clock object
YRealTimeClock

realtimeclock\rightarrow get\_errorType()
realtimeclock\rightarrow errorType()
realtimeclock.get\_errorType()

Returns the numerical error code of the latest error with the clock.

int get\_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns**:

a number corresponding to the code of the latest error that occurred while using the clock object
YRealTimeClock

realtimeclock→get_friendlyName()
realtimeclock→friendlyName()
realtimeclock.get_friendlyName()

Returns a global identifier of the clock in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName( )

The returned string uses the logical names of the module and of the clock if they are defined, otherwise the serial number of the module and the hardware identifier of the clock (for example: MyCustomName.relay1)

Returns :
    a string that uniquely identifies the clock using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
YRealTimeClock
realtimeclock→get_functionDescriptor()
realtimeclock→functionDescriptor()
realtimeclock.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
3. Reference

**realtimeclock**→**get_functionId()**

**realtimeclock**→**functionId()**

**realtimeclock.get_functionId()**

Returns the hardware identifier of the clock, without reference to the module.

`String get_functionId()`

For example `relay1`

**Returns:**

- A string that identifies the clock (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
realtimeclock→get_hardwareId()
realtimeclock→hardwareId()
realtimeclock.get_hardwareId()

Returns the unique hardware identifier of the clock in the form SERIAL.FUNCTIONID.

String get_hardwareId()  
The unique hardware identifier is composed of the device serial number and of the hardware identifier of the clock (for example RELAYLO1-123456.relay1).

Returns:
- a string that uniquely identifies the clock (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
Returns the logical name of the clock.

String `get_logicalName()`

**Returns:**

- a string corresponding to the logical name of the clock.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
realtimeclock→get_module()
realtimeclock→module()
realtimeclock.get_module()

Gets the YModule object for the device on which the function is located.

YModule get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns:

an instance of YModule
realtimeclock\rightarrow get\_timeSet()
realtimeclock\rightarrow timeSet()
realtimeclock\cdot get\_timeSet()

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`.
realtimeclock\rightarrow\texttt{get\_unixTime()}

realtimeclock\rightarrow\texttt{unixTime()}

\texttt{realtimeclock.get\_unixTime()}

\begin{quote}
Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).
\end{quote}

\begin{verbatim}
long \texttt{get\_unixTime( )}
\end{verbatim}

\begin{quote}
\textbf{Returns :}
\begin{itemize}
\item an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)
\end{itemize}

On failure, throws an exception or returns \texttt{Y\_UNIXTIME\_INVALID}.
\end{quote}
Returns the value of the userData attribute, as previously stored using method `set_userData`.

**Returns:**
the object stored previously by the caller.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>realtimeclock.get_utcOffset()</code></td>
<td>Returns the number of seconds between current time and UTC time (time zone).</td>
</tr>
<tr>
<td><code>realtimeclock.utcOffset()</code></td>
<td>Returns : an integer corresponding to the number of seconds between current time and UTC time (time zone)</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_UTCOFFSET_INVALID.</td>
</tr>
</tbody>
</table>
realtimeclock→isOnline()  
realtimeclock.isOnline()  

YRealTimeClock

Checks if the clock is currently reachable, without raising any error.

boolean isOnline()

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

Returns:

true if the clock can be reached, and false otherwise
realtimeclock→load() \texttt{realtimeclock.load()}

Preloads the clock cache with a specified validity duration.

\textbf{int load( long msValidity)}

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

\begin{itemize}
  \item \textbf{Parameters :}
    \begin{itemize}
      \item \textbf{msValidity} an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds
    \end{itemize}
  \item \textbf{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.
3. Reference

realtimeclock\rightarrow nextRealTimeClock()

YRealTimeClock

realtimeclock.nextRealTimeClock()

Continues the enumeration of clocks started using \texttt{yFirstRealTimeClock()}.

\begin{tabular}{|p{10cm}|}
\hline
YRealTimeClock \texttt{nextRealTimeClock()} \\
\hline
\begin{itemize}
  \item \textbf{Returns :}
  \begin{itemize}
    \item a pointer to a \texttt{YRealTimeClock} object, corresponding to a clock currently online, or a \texttt{null} pointer if there are no more clocks to enumerate.
  \end{itemize}
\end{itemize}
\end{tabular}
realtimeclock→registerValueCallback()  YRealTimeClock
realtimeclock.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

- **callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
3. Reference

### realtimeclock → set_logicalName()
### realtimeclock → setLogicalName()
### realtimeclock.set_logicalName()

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.

### 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.
3. Reference

realtimeclock → set_userData()  
realtimeclock → setUserData()  
realtimeclock.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 number of seconds between current time and UTC time (time zone).

```c
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:

```javascript
<script type='text/javascript' src='yocto_refframe.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YRefFrame = yoctolib.YRefFrame;
```
```php
require_once('yocto_refframe.php');
```
```cpp
#include "yocto_refframe.h"
```
```m
uses yocto_refframe;
```
```pas
uses yocto_refframe;
```
```vb
yocto_refframe.vb
```
```cs
yocto_refframe.cs
```
```java
import com.yoctopuce.YoctoAPI.YRefFrame;
```
```py
from yocto_refframe import *
```

### Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindRefFrame(func)</code></td>
<td>Retrieves a reference frame for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstRefFrame()</code></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><code>refframe→cancel3DCalibration()</code></td>
<td>Aborts the sensors tridimensional calibration process and restores normal settings.</td>
</tr>
<tr>
<td><code>refframe→describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the reference frame in the form <code>TYPE(NAME)=SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><code>refframe→get_3DCalibrationHint()</code></td>
<td>Returns instructions to proceed to the tridimensional calibration initiated with method <code>start3DCalibration</code>.</td>
</tr>
<tr>
<td><code>refframe→get_3DCalibrationLogMsg()</code></td>
<td>Returns the latest log message from the calibration process.</td>
</tr>
<tr>
<td><code>refframe→get_3DCalibrationProgress()</code></td>
<td>Returns the global process indicator for the tridimensional calibration initiated with method <code>start3DCalibration</code>.</td>
</tr>
<tr>
<td><code>refframe→get_3DCalibrationStage()</code></td>
<td>Returns index of the current stage of the calibration initiated with method <code>start3DCalibration</code>.</td>
</tr>
<tr>
<td><code>refframe→get_3DCalibrationStageProgress()</code></td>
<td>Returns the process indicator for the current stage of the calibration initiated with method <code>start3DCalibration</code>.</td>
</tr>
<tr>
<td><code>refframe→get_advertisedValue()</code></td>
<td>Returns the current value of the reference frame (no more than 6 characters).</td>
</tr>
<tr>
<td><code>refframe→get_bearing()</code></td>
<td>Returns the reference bearing used by the compass.</td>
</tr>
</tbody>
</table>
reframe→get_errorMessage()
Returns the error message of the latest error with the reference frame.

reframe→get_errorType()
Returns the numerical error code of the latest error with the reference frame.

reframe→get_friendlyName()
Returns a global identifier of the reference frame in the format MODULE_NAME.FUNCTION_NAME.

reframe→get_functionDescriptor()
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

reframe→get_functionId()
Returns the hardware identifier of the reference frame, without reference to the module.

reframe→get_hardwareId()
Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

reframe→get_logicalName()
Returns the logical name of the reference frame.

reframe→get_module()
Gets the YModule object for the device on which the function is located.

reframe→get_module_async(callback, context)
Gets the YModule object for the device on which the function is located (asynchronous version).

reframe→get_mountOrientation()
Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_mountPosition()
Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_userData()
Returns the value of the userData attribute, as previously stored using method set_userData.

reframe→isOnline()
Checks if the reference frame is currently reachable, without raising any error.

reframe→isOnline_async(callback, context)
Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reframe→load(msValidity)
Preloads the reference frame cache with a specified validity duration.

reframe→load_async(msValidity, callback, context)
Preloads the reference frame cache with a specified validity duration (asynchronous version).

reframe→more3DCalibration()
Continues the sensors tridimensional calibration process previously initiated using method start3DCalibration.

reframe→nextRefFrame()
Continues the enumeration of reference frames started using yFirstRefFrame().

reframe→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

reframe→save3DCalibration()
Applies the sensors tridimensional calibration parameters that have just been computed.

reframe→set_bearing(newval)
Changes the reference bearing used by the compass.

reframe→set_logicalName(newval)
Changes the logical name of the reference frame.

```
refframe→set_mountPosition(position, orientation)
```

Changes the compass and tilt sensor frame of reference.

```
refframe→set_userData(data)
```

Stores a user context provided as argument in the userData attribute of the function.

```
refframe→start3DCalibration()
```

Initiates the sensors tridimensional calibration process.

```
refframe→wait_async(callback, context)
```

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YRefFrame.FindRefFrame()

YRefFrame.FindRefFrame(String func)

Retrieves a reference frame 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 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.

Use the method YRefFrame.nextRefFrame() to iterate on next reference frames.

**Returns:**

a pointer to a YRefFrame object, corresponding to the first reference frame currently online, or a null pointer if there are none.
refframe->cancel3DCalibration()

crefframe.cancel3DCalibration()

Aborts the sensors tridimensional calibration process and restores normal settings.

int cancel3DCalibration()

On failure, throws an exception or returns a negative error code.
YRefFrame

refframe→describe()refframe.describe() 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.
3. Reference

YRefFrame

refframe→get_3DCalibrationLogMsg()
refframe→3DCalibrationLogMsg()
refframe.get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

String get_3DCalibrationLogMsg()

When no new message is available, returns an empty string.

Returns:
a character string.
3. Reference

YRefFrame

`reframe.get_3DCalibrationProgress()`

`reframe.get_3DCalibrationProgress()`

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

```c
int get_3DCalibrationProgress()
```

**Returns:**

an integer between 0 (not started) and 100 (stage completed).
Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

```c
int get_3DCalibrationStage() {
    // Implementation details...
    return currentStage; // Replace with actual stage index calculation
}
```

**Returns:**
- an integer, growing each time a calibration stage is completed.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YRefFrame.get_3DCalibrationStageProgress()</td>
<td>Returns the process indicator for the current stage of the calibration initiated with method start3DCalibration.</td>
</tr>
<tr>
<td>int get_3DCalibrationStageProgress()</td>
<td>Returns: an integer between 0 (not started) and 100 (stage completed).</td>
</tr>
</tbody>
</table>
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`. 
reframe->get_bearing()

Returns the reference bearing used by the compass.

double get_bearing()

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

**Returns:**
- a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns Y_BEARING_INVALID.
YRefFrame

refframe→get_errorMessage()
refframe→errorMessage()
refframe.getMessage()

Returns the error message of the latest error with the reference frame.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
- a string corresponding to the latest error message that occurred while using the reference frame object
Get the numerical error code of the latest error with the reference frame.

```cpp
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**
- a number corresponding to the code of the latest error that occurred while using the reference frame object.
3. Reference

YRefFrame

refframe→get_friendlyName()  
refframe→friendlyName()  
refframe.get_friendlyName()

Returns a global identifier of the reference frame in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName( )

The returned string uses the logical names of the module and of the reference frame if they are defined, otherwise the serial number of the module and the hardware identifier of the reference frame (for example: MyCustomName.relay1)

Returns:

- a string that uniquely identifies the reference frame using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
YRefFrame

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```java
String get_functionDescriptor() {
    // Implementation...
}
```

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`.***
reframe->get_functionId()
refframe->functionId()
refframe.get_functionId()

YRefFrame

Returns the hardware identifier of the reference frame, without reference to the module.

String get_functionId()

For example relay1

Returns :

a string that identifies the reference frame (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
YRefFrame

YRefFrame.get_hardwareId()

YRefFrame.hardwareId()

refframe.get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame (for example RELAYLO1-123456.relay1).

**String get_hardwareId()**

Returns:
- A string that uniquely identifies the reference frame (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
Returns the logical name of the reference frame.

**String get_logicalName()**

**Returns:**

a string corresponding to the logical name of the reference frame.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
refframe→get_module()  
refframe→module()refframe.get_module()  

YRefFrame  

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
`refframe→get_mountOrientation()`

`refframe→mountOrientation()`

`refframe.get_mountOrientation()`

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

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.

MOUNTPOSITION get_mountPosition()

Returns:

- a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.
### 3. Reference

**YRefFrame**

`refframe.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.

**Object get_userData()**

- **Returns**: the object stored previously by the caller.
YRefFrame

reframe.isOnline()

Checks if the reference frame is currently reachable, without raising any error.

boolean isOnline( )

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

Returns :

ture if the reference frame can be reached, and false otherwise
Preloads the reference frame cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.
### Reference

```cpp
reframe
-> nextRefFrame()
```

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

```cpp
YRefFrame
nextRefFrame()
```

**Returns:**

A pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a null pointer if there are no more reference frames to enumerate.
reframe/registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
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.

```java
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.
Changes the compass and tilt sensor frame of reference.

```
int set_mountPosition( MOUNTPOSITION position, MOUNTORIENTATION orientation)
```

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

**Parameters:**

- **position**
  
  A value among the `Y_MOUNTPOSITION` enumeration (`Y_MOUNTPOSITION_BOTTOM`, `Y_MOUNTPOSITION_TOP`, `Y_MOUNTPOSITION_FRONT`, `Y_MOUNTPOSITION_RIGHT`, `Y_MOUNTPOSITION_REAR`, `Y_MOUNTPOSITION_LEFT`), corresponding to the installation in a box, on one of the six faces.

- **orientation**

  A value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`, `Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the `saveToFlash()` method of the module if the modification must be kept.
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
YRefFrame

YRefFrame.start3DCalibration()

Initiates the sensors tridimensional calibration process.

int start3DCalibration()

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method get_3DCalibrationHint, and more3DCalibration should be invoked about 5 times per second. The calibration procedure is completed when the method get_3DCalibrationProgress returns 100. At this point, the computed calibration parameters can be applied using method save3DCalibration. The calibration process can be canceled at any time using method cancel3DCalibration. On failure, throws an exception or returns a negative error code.
3. Reference

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"
```

```pas
uses yocto Relay;
```

```vb
yocto_relay.vb
```

```cs
yocto_relay.cs
```

```java
import com.yoctopuce.YoctoAPI.YRelay;
```

```py
from yocto_relay import *
```

### Global functions

**yFindRelay(func)**

Retrieves a relay for a given identifier.

**yFirstRelay()**

Starts the enumeration of relays currently accessible.

### YRelay methods

**relay→delayedPulse(ms_delay, ms_duration)**

Schedules a pulse.

**relay→describe()**

Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**relay→get_advertisedValue()**

Returns the current value of the relay (no more than 6 characters).

**relay→get_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

**relay→get_errorMessage()**

Returns the error message of the latest error with the relay.

**relay→get_errorType()**

Returns the numerical error code of the latest error with the relay.

**relay→get_friendlyName()**

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

**relay→get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**relay→get_functionId()**

Returns the hardware identifier of the relay, without reference to the module.

**relay→get_hardwareId()**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_logicalName()</code></td>
<td>Returns the logical name of the relay.</td>
</tr>
<tr>
<td><code>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>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>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_output()</code></td>
<td>Returns the output state of the relays, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td><code>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>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>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>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 relay is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>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>load(msValidity)</code></td>
<td>Preloads the relay cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>load_async(msValidity, callback, context)</code></td>
<td>Preloads the relay cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>nextRelay()</code></td>
<td>Continues the enumeration of relays started using <code>yFirstRelay()</code>.</td>
</tr>
<tr>
<td><code>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>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 relay.</td>
</tr>
<tr>
<td><code>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>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>relay→set_output(newval)</td>
<td>Changes the output state of the relays, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td>relay→set_state(newval)</td>
<td>Changes the state of the relays (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td>relay→set_stateAtPowerOn(newval)</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>relay→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td>relay→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>
YRelay.FindRelay()

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()

Starts the enumeration of relays currently accessible.

YRelay.FirstRelay()

Use the method YRelay.nextRelay() to iterate on next relays.

Returns:
- a pointer to a YRelay object, corresponding to the first relay currently online, or a null pointer if there are none.
relay\rightarrow \text{delayedPulse()}

Schedules a pulse.

\begin{verbatim}
int delayedPulse( int ms_delay, int ms_duration)
\end{verbatim}

**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.
3. Reference

relay\rightarrow\text{describe()} \quad \text{YRelay}

Returns a short text that describes unambiguously the instance of the relay in the form
\text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID}.

\begin{verbatim}
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 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 :
\text{a string that describes the relay (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)}
\end{verbatim}
Returns the current value of the relay (no more than 6 characters).

String get_advertisedValue()  

Returns:

   a string corresponding to the current value of the relay (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
3. Reference

**YRelay**

relay→get_countdown()  
relay→countdown()  
relay.get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

```java
long get_countdown() {
    Returns:
    an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

    On failure, throws an exception or returns Y_COUNTDOWN_INVALID.
```

relay→get_errorMessage()
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
Returns the numerical error code of the latest error with the relay.

```cpp
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the relay object
```java
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.
### 3. Reference

**relay**→**get_functionDescriptor**()  
**YRelay**  
**relay**→**functionDescriptor**()  
**relay.get_functionDescriptor**()  

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```java
String get_functionDescriptor()
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**  
an identifier of type `YFUN_DESCR`.  

**If the function has never been contacted, the returned value is**  
`Y_FUNCTION DESCRIPTOR INVALID`.  

---
3. Reference

**YRelay**

`relay→get_functionId()`

`relay→functionId()`

`relay.get_functionId()`

Returns the hardware identifier of the relay, without reference to the module.

**String get_functionId( )**

For example `relay1`

**Returns :**

- a string that identifies the relay (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
3. Reference

`relay.get_hardwareId()`

Returns the unique hardware identifier of the relay 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 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`. 
YRelay

relay→get_logicalName()
relay→logicalName()relay.get_logicalName()

Returns the logical name of the relay.

String get_logicalName()

Returns:

- a string corresponding to the logical name of the relay.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
3. Reference

YRelay

relay→get_maxTimeOnStateA()
relay→maxTimeOnStateA()
relay.get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

long get_maxTimeOnStateA( )

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.
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 Y_MAXTIMEONSTATEB_INVALID.
3. Reference

YRelay

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
YRelay

relay→get_output()
relay→output() relay.get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

int get_output( )

Returns:
- either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.
### 3. Reference

<table>
<thead>
<tr>
<th>YRelay</th>
</tr>
</thead>
</table>

```c
relay->get_pulseTimer()
relay->pulseTimer() relay.get_pulseTimer()
```

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

#### long get_pulseTimer( )

When there is no ongoing pulse, returns zero.

**Returns:**
- An integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns **Y_PULSETIMER_INVALID**.
Returns the state of the relays (A for the idle position, B for the active position).

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).

```c
int get_stateAtPowerOn()
```

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>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)</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.
get_userData() → relay.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

Object `get_userData()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns**:

the object stored previously by the caller.
relay→isOnline() \( \text{relay.isOnline()} \)

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.

```c
int load(long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
3. Reference

**YRelay**

```cpp
YRelay nextRelay()
```

Continues the enumeration of relays started using `yFirstRelay()`.

**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).

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.
YRelay

3. Reference

relay→registerValueCallback()
relay.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
Changes the logical name of the relay.

```java
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.
3. Reference

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

```
int set_maxTimeOnStateA( long newval)
```

Use zero for no maximum time.

**Parameters :**
- `newval` an integer

**Returns :**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
relay->set_maxTimeOnStateB()
relay->setMaxTimeOnStateB()
relay.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```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.
3. Reference

relay->set_output()
relay->setOutput() relay.set_output()

Changes the output state of the relays, when used as a simple switch (single throw).

| int set_output( int newval) |

**Parameters:**

- **newval** either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

**Returns:**

- YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
3. Reference

YRelay

relay.set_state()

Changes the state of the relays (A for the idle position, B for the active position).

```
int set_state( int newval)
```

**Parameters:**

- `newval` either `Y_STATE_A` or `Y_STATE_B`, according to the state of the relays (A for the idle position, B for the active position)

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
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.

```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. Reference

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;
```

```php
require_once('yocto_api.php');
```

```cpp
#include "yocto_api.h"
```

```m
#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(func)**

Retrieves a sensor for a given identifier.

**yFirstSensor()**

Starts the enumeration of sensors currently accessible.

### YSensor methods

**sensor→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**sensor→describe()**

Returns a short text that describes unambiguously the instance of the sensor in the form

TYPE(NAME)=SERIAL.FUNCTIONID.

**sensor→get_advertisedValue()**

Returns the current value of the sensor (no more than 6 characters).

**sensor→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, 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)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sensor.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>sensor.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>sensor.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>sensor.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>sensor.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>
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.
3. Reference

YSensor.FirstSensor()

yFirstSensor() YSensor.FirstSensor()

Starts the enumeration of sensors currently accessible.

YSensor FirstSensor()

Use the method YSensor.nextSensor() to iterate on next sensors.

Returns:
- a pointer to a YSensor object, corresponding to the first sensor currently online, or a null pointer if there are none.
YSensor

sensor → calibrateFromPoints()

sensor.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

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.
senson.describe() 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 it the name used for the first access to the
function, SERIAL is the serial number of the module if the module is connected or "unresolved", and
FUNCTIONID is the hardware identifier of the function if the module is connected. For example,
this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the
module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module
has not yet been connected. This method does not trigger any USB or TCP transaction and can
therefore be used in a debugger.

Returns:
a string that describes the sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-
123456.relay1)
sensor→get_advertisedValue()
sensor->advertisedValue()
sensor.get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

String get_advertisedValue( )

**Returns:**
- a string corresponding to the current value of the sensor (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
sensor→get_currentRawValue()
sensor→currentRawValue()
sensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

Returns:

- a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
sensor→get_currentValue()
sensor→currentValue()
sensor.get_currentValue()

Returns the current value of the measure, in the specified unit, as a floating point number.

```java
double get_currentValue()```

**Returns:**

- a floating point number corresponding to the current value of the measure, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
3. Reference

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.

int get_errorType( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:

a number corresponding to the code of the latest error that occurred while using the sensor object
sensor→get_friendlyName()  sensor→friendlyName()  sensor.get_friendlyName()

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\rightarrow get\_functionDescriptor()
sensor\rightarrow functionDescriptor()
sensor.get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

String get\_functionDescriptor( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN\_DESCR.

If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.
sensor→get_functionId()
sensor→functionId()sensor.get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

String get_functionId()

For example relay1

**Returns:**

- a string that identifies the sensor (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
sensor→get_hardwareId()
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.
Returns the maximal value observed for the measure since the device was started.

**double get_highestValue()**

**Returns:**

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.
sensor→get_logFrequency()
sensor→logFrequency()
sensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String get_logFrequency()

Returns:

- a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.
Returns the logical name of the sensor.

String `get_logicalName()`

**Returns:**
- a string corresponding to the logical name of the sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
Returns the minimal value observed for the measure since the device was started.

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a floating point number corresponding to the minimal value observed for the measure since the device was started</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.
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.

**YSensor**

```java
sensor.get_recordedData()
```

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.

```java
YDataSet get_recordedData( long startTime, long endTime)
```

**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

sensor->get_reportFrequency()

sensor->reportFrequency()

sensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

<table>
<thead>
<tr>
<th>String get_reportFrequency()</th>
</tr>
</thead>
</table>

**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()**

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 measure.

**String get_unit()**

**Returns:**
- a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.
sensor → get_userData()  

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.

**Returns:**
- `true` if the sensor can be reached, and `false` otherwise.
sensor.load() \rightarrow \text{sensor.load}()

Preloads the sensor cache with a specified validity duration.

\textbf{int load}( \text{long msValidity})

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- \textbf{msValidity} an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- \text{YAPI_SUCCESS} when the call succeeds.

On failure, throws an exception or returns a negative error code.
### Load Calibration Points

**Sensor Method**

```java
sensor.loadCalibrationPoints()
```

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

#### 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.

```java
int loadCalibrationPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```
sensor.nextSensor() \( \rightarrow \) nextSensor()

Continues the enumeration of sensors started using \( \text{yFirstSensor}() \).

\( \text{YSensor} \) nextSensor()

**Returns:**

- a pointer to a \( \text{YSensor} \) 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.

```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.
sensor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
sensor→set_highestValue()
sensor→setHighestValue()
sensor.set_highestValue()

Changes the recorded maximal value observed.

int set_highestValue( double newval)

Parameters :
  - newval a floating point number corresponding to the recorded maximal value observed

Returns :
  YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.
sensor\rightarrow\text{set\_logFrequency()}

Changes the datalogger recording frequency for this function.

```c
int \text{set\_logFrequency}( \text{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.
### 3. Reference

sensor->set_logicalName()
sensor->setLogicalName()
sensor.set_logicalName()

Changes the logical name of the sensor.

<table>
<thead>
<tr>
<th>int set_logicalName(String newval)</th>
</tr>
</thead>
</table>

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.

```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.
3. Reference

sensor→set_reportFrequency()
sensor→setReportFrequency()
sensor.set_reportFrequency()

Changes the timed value notification frequency for this function.

int set_reportFrequency( String newval)

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters:**
- **newval** a string corresponding to the timel 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>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code> a floating point number corresponding to the resolution of the measured physical values</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.
sensor→set_userData()  
sensor→setUserData()  
sensor.set_userData()  

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
3.40. 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>
```

```nodejs
var yoctolib = require('yoctolib');
var YSerialPort = yoctolib.YSerialPort;
require_once('yocto_serialport.php');
#include "yocto_serialport.h"
#include "yocto_serialport.h"
uses yocto_serialport;
yocto_serialport.vb
yocto_serialport.cs
yocto_serialport.js
yocto_serialport.php
```

### 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()**
3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serialport→<strong>get_logicalName</strong>()</td>
<td>Returns the logical name of the serial port.</td>
</tr>
<tr>
<td>serialport→<strong>get_module</strong>()</td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td>serialport→<strong>get_module_async</strong>(callback, context)</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td>serialport→<strong>get_msgCount</strong>()</td>
<td>Returns the total number of messages received since last reset.</td>
</tr>
<tr>
<td>serialport→<strong>get_protocol</strong>()</td>
<td>Returns the type of protocol used over the serial line, as a string.</td>
</tr>
<tr>
<td>serialport→<strong>get_rxCount</strong>()</td>
<td>Returns the total number of bytes received since last reset.</td>
</tr>
<tr>
<td>serialport→<strong>get_serialMode</strong>()</td>
<td>Returns the serial port communication parameters, as a string such as “9600,8N1”.</td>
</tr>
<tr>
<td>serialport→<strong>get_txCount</strong>()</td>
<td>Returns the total number of bytes transmitted since last reset.</td>
</tr>
<tr>
<td>serialport→<strong>get_userData</strong>()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td>serialport→<strong>isOnline</strong>()</td>
<td>Checks if the serial port is currently reachable, without raising any error.</td>
</tr>
<tr>
<td>serialport→<strong>isOnline_async</strong>(callback, context)</td>
<td>Checks if the serial port is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td>serialport→<strong>load</strong>(msValidity)</td>
<td>Preloads the serial port cache with a specified validity duration.</td>
</tr>
<tr>
<td>serialport→<strong>load_async</strong>(msValidity, callback, context)</td>
<td>Preloads the serial port cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td>serialport→<strong>modbusReadBits</strong>(slaveNo, pduAddr, nBits)</td>
<td>Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.</td>
</tr>
<tr>
<td>serialport→<strong>modbusReadInputBits</strong>(slaveNo, pduAddr, nBits)</td>
<td>Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.</td>
</tr>
<tr>
<td>serialport→<strong>modbusReadInputRegisters</strong>(slaveNo, pduAddr, nWords)</td>
<td>Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.</td>
</tr>
<tr>
<td>serialport→<strong>modbusReadRegisters</strong>(slaveNo, pduAddr, nWords)</td>
<td>Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.</td>
</tr>
<tr>
<td>serialport→<strong>modbusWriteAndReadRegisters</strong>(slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)</td>
<td>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.</td>
</tr>
<tr>
<td>serialport→<strong>modbusWriteBit</strong>(slaveNo, pduAddr, value)</td>
<td>Sets a single internal bit (or coil) on a MODBUS serial device.</td>
</tr>
<tr>
<td>serialport→<strong>modbusWriteBits</strong>(slaveNo, pduAddr, bits)</td>
<td>Sets several contiguous internal bits (or coils) on a MODBUS serial device.</td>
</tr>
<tr>
<td>serialport→<strong>modbusWriteRegister</strong>(slaveNo, pduAddr, value)</td>
<td>Sets a single internal register (or holding register) on a MODBUS serial device.</td>
</tr>
</tbody>
</table>
| serialport→**modbusWriteRegisters**(slaveNo, pduAddr, values) | }
Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

```plaintext
serialport→nextSerialPort()
Continues the enumeration of serial ports started using yFirstSerialPort().
```

Sends a text line query to the serial port, and reads the reply, if any.

```plaintext
serialport→queryLine(query, maxWait)
```

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

```plaintext
serialport→queryMODBUS(slaveNo, pduBytes)
```

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

```plaintext
serialport→readHex(nBytes)
```

Reads a single line (or message) from the receive buffer, starting at current stream position.

```plaintext
serialport→readLine()
```

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

```plaintext
serialport→readMessages(pattern, maxWait)
```

Reads data from the receive buffer as a string, starting at current stream position.

```plaintext
serialport→readStr(nChars)
```

Changes the current internal stream position to the specified value.

```plaintext
serialport→read_seek(rxCountVal)
```

Registers the callback function that is invoked on every change of advertised value.

```plaintext
serialport→registerValueCallback(callback)
```

Clears the serial port buffer and resets counters to zero.

```plaintext
serialport→reset()
```

Manually sets the state of the RTS line.

```plaintext
serialport→set_RTS(val)
```

Changes the logical name of the serial port.

```plaintext
serialport→set_logicalName(newval)
```

Changes the type of protocol used over the serial line.

```plaintext
serialport→set_protocol(newval)
```

Changes the serial port communication parameters, with a string such as "9600,8N1".

```plaintext
serialport→set_serialMode(newval)
```

Stores a user context provided as argument in the userData attribute of the function.

```plaintext
serialport→set_userData(data)
```

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```plaintext
serialport→wait_async(callback, context)
```

Sends a byte sequence (provided as a list of bytes) to the serial port.

```plaintext
serialport→writeArray(byteList)
```

Sends a binary buffer to the serial port, as is.

```plaintext
serialport→writeBin(buff)
```

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

```plaintext
serialport→writeHex(hexString)
```

Sends an ASCII string to the serial port, followed by a line break (CR LF).

```plaintext
serialport→writeLine(text)
```

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

```plaintext
serialport→writeMODBUS(hexString)
```

serialport→writeStr(text)
| Sends an ASCII string to the serial port, as is. |
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()

Starts the enumeration of serial ports currently accessible.

YSerialPort FirstSerialPort()

Use the method YSerialPort.nextSerialPort() to iterate on next serial ports.

Returns:

- a pointer to a YSerialPort object, corresponding to the first serial port currently online, or a null pointer if there are none.
### YSerialPort

**serialport.describe()**

Returns a short text that describes unambiguously the instance of the serial 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(BadCustomerName.relay1)=unresolved** if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the serial port (ex: **Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1**)
serialport→get_CTS()
serialport→CTS() serialport.get_CTS()

Read the level of the CTS line.

```c
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.
serialport → get_advertisedValue()

serialport → advertisedValue()

serialport.get_advertisedValue()

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.
Returns the total number of communication errors detected since last reset.

`int get_errCount()`

**Returns:**
- an integer corresponding to the total number of communication errors detected since last reset

On failure, throws an exception or returns `Y_ERRCOUNT_INVALID`. 

```java
YSerialPort serialport
serialport→errCount()
serialport.get_errCount()
```
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
Returns the numerical error code of the latest error with the serial port.

```c
int get_errorType() 
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

- a number corresponding to the code of the latest error that occurred while using the serial port object
`serialport.get_friendlyName()` returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

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`. 
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor() This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
serialport→get_functionId()
serialport→functionId()
serialport.get_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

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.MODELID`.

**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`. 

---

YSerialPort

serialport.get_hardwareId()
<table>
<thead>
<tr>
<th>serialport → get_lastMsg()</th>
<th>YSerialPort</th>
</tr>
</thead>
<tbody>
<tr>
<td>serialport → lastMsg()</td>
<td>serialport.get_lastMsg()</td>
</tr>
</tbody>
</table>

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()

<table>
<thead>
<tr>
<th>Gets the YModule object for the device on which the function is located.</th>
</tr>
</thead>
<tbody>
<tr>
<td>YModule get_module( )</td>
</tr>
<tr>
<td>If the function cannot be located on any module, the returned instance of YModule is not shown as online.</td>
</tr>
<tr>
<td>Returns:</td>
</tr>
<tr>
<td>an instance of YModule</td>
</tr>
<tr>
<td>serialport → get_msgCount()</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>serialport → msgCount()</td>
</tr>
<tr>
<td>serialport.get_msgCount()</td>
</tr>
</tbody>
</table>

Returns the total number of messages received since last reset.

```c
int get_msgCount();
```

**Returns:**
- an integer corresponding to the total number of messages received since last reset

On failure, throws an exception or returns `Y_MSGCOUNT_INVALID`. 

serialport→get_protocol()

serialport→protocol()serialport.get_protocol()

YSerialPort

Returns the type of protocol used over the serial line, as a string.

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.
3. Reference

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.
3. Reference

YSerialPort

serialport→get_serialMode()
serialport->serialMode()
serialport.get_serialMode()

Returns the serial port communication parameters, as a string such as "9600,8N1".

String get_serialMode()  
The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix is included if flow control is active: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Returns :
  a string corresponding to the serial port communication parameters, as a string such as "9600,8N1"

On failure, throws an exception or returns Y_SERIALMODE_INVALID.
YSerialPort

3. Reference

serialport \rightarrow get\_txCount()

serialport \rightarrow txCount()
serialport.get\_txCount()

<table>
<thead>
<tr>
<th>Returns</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the total number of bytes transmitted since last reset.</td>
<td></td>
</tr>
</tbody>
</table>

```
int get\_txCount()
```

**Returns:**
- an integer corresponding to the total number of bytes transmitted since last reset

On failure, throws an exception or returns Y\_TXCOUNT\_INVALID.
serialport→getUserData()
serialport→userData()
serialport.getUserData()

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.
3. Reference

serialport→isOnline()serialport.isOnline()  
YSerialPort

Checks if the serial port is currently reachable, without raising any error.

boolean isOnline()

If there is a cached value for the serial port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the serial port.

Returns:
true if the serial port can be reached, and false otherwise
### serialport.load()

Preloads the serial port cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

```java
ArrayList<Integer> 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.

ArrayList<Integer> 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.

```java
ArrayList<Integer> 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.
serialport→modbusReadRegisters()

serialport.modbusReadRegisters()

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

ArrayList<Integer> 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.
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.

```java
ArrayList<Integer> modbusWriteAndReadRegisters( int slaveNo, int pduWriteAddr, ArrayList<Integer> 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.
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.
SerialPort

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

```java
int modbusWriteBits(int slaveNo, int pduAddr, ArrayList<Integer> bits)
```

This method uses the MODBUS function code 0x0f (Write Multiple Coils).

**Parameters:**
- `slaveNo` the address of the slave MODBUS device to drive
- `pduAddr` the relative address of the first bit/coil to set (zero-based)
- `bits` the vector of bits to be set (one integer per bit)

**Returns:**
the number of bits/coils affected on the device

On failure, throws an exception or returns zero.
serialport → modbusWriteRegister()  
serialport.modbusWriteRegister()

Sets a single internal register (or holding register) on a MODBUS serial device.

```c
int modbusWriteRegister( int slaveNo, int pduAddr, int 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.
### 3. Reference

```java
serialport → modbusWriteRegisters()
serialport.modbusWriteRegisters()
```

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

```java
int modbusWriteRegisters(
    int slaveNo,
    int pduAddr,
    ArrayList<Integer> 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→nextSerialPort()  
YSerialPort

serialport.nextSerialPort()  

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.
### YSerialPort

**serialport → queryLine()**  

Sends a text line query to the serial port, and reads the reply, if any.

**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.
serialport→queryMODBUS()

serialport.queryMODBUS()

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

```java
ArrayList<Integer> queryMODBUS(int slaveNo,
                                ArrayList<Integer> 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()

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, e.g. '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.

ArrayList<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.
serialport→readStr() \( \text{serialport.readStr()} \)

Reads data from the receive buffer as a string, starting at current stream position.

\[ \text{String readStr( int nChars)} \]

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.
Changes the current internal stream position to the specified value.

```c
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\text{registerValueCallback()}

\text{serialport.registerValueCallback()}

Registers the callback function that is invoked on every change of advertised value.

\text{int registerValueCallback( UpdateCallback \ 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 \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.
\end{itemize}
3. Reference

serialport.reset() serialport.reset()

Cleans the serial port buffer and resets counters to zero.

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.

```cpp
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.

```java
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.

```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
serialport → writeArray()  \[\text{serialport.writeArray()}\]  \[\text{YSerialPort}\]

Sends a byte sequence (provided as a list of bytes) to the serial port.

```java
int writeArray( ArrayList<Integer> byteList)
```

**Parameters:**
- `byteList` a list of byte codes

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
`serialport.writeBin()` sends a binary buffer to the serial port, as is.

```c
int writeBin( )
```

**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.
### YSerialPort

**serialport.writeHex()**

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

```java
int writeHex(String hexString)
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hexString</code></td>
<td>a string of hexadecimal byte codes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS</td>
<td>if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
serialport.writeLine() sends an ASCII string to the serial port, followed by a line break (CR LF).

```java
int writeLine(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.
Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

```java
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.

```cpp
int writeStr(const 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 a same move.

In order to use the functions described here, you should include:

```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>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>servo-&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>servo-&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>servo-&gt;get_neutral()</code></td>
<td>Returns the duration in microseconds of a neutral pulse for the servo.</td>
</tr>
<tr>
<td><code>servo-&gt;get_position()</code></td>
<td>Returns the current servo position.</td>
</tr>
<tr>
<td><code>servo-&gt;get_positionAtPowerOn()</code></td>
<td>Returns the servo position at device power up.</td>
</tr>
<tr>
<td><code>servo-&gt;get_range()</code></td>
<td>Returns the current range of use of the servo.</td>
</tr>
<tr>
<td><code>servo-&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>servo-&gt;isOnline()</code></td>
<td>Checks if the servo is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>servo-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the servo is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>servo-&gt;load(msValidity)</code></td>
<td>Preloads the servo cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>servo-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the servo cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>servo-&gt;move(target, ms_duration)</code></td>
<td>Performs a smooth move at constant speed toward a given position.</td>
</tr>
<tr>
<td><code>servo-&gt;nextServo()</code></td>
<td>Continues the enumeration of servos started using <code>yFirstServo()</code>.</td>
</tr>
<tr>
<td><code>servo-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><code>servo-&gt;set_enabled(newval)</code></td>
<td>Stops or starts the servo.</td>
</tr>
<tr>
<td><code>servo-&gt;set_enabledAtPowerOn(newval)</code></td>
<td>Configure the servo signal generator state at power up.</td>
</tr>
<tr>
<td><code>servo-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the servo.</td>
</tr>
<tr>
<td><code>servo-&gt;set_neutral(newval)</code></td>
<td>Changes the duration of the pulse corresponding to the neutral position of the servo.</td>
</tr>
<tr>
<td><code>servo-&gt;set_position(newval)</code></td>
<td>Changes immediately the servo driving position.</td>
</tr>
<tr>
<td><code>servo-&gt;set_positionAtPowerOn(newval)</code></td>
<td>Configure the servo position at device power up.</td>
</tr>
<tr>
<td><code>servo-&gt;set_range(newval)</code></td>
<td>Changes the range of use of the servo, specified in per cents.</td>
</tr>
<tr>
<td><code>servo-&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>servo-&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.
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.

YServo FirstServo( )

Use the method YServo.nextServo() to iterate on next servos.

**Returns:**
- a pointer to a YServo object, corresponding to the first servo currently online, or a null pointer if there are none.
servo.describe()  

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` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:

a string that describes the servo (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
3. Reference

Returns the current value of the servo (no more than 6 characters).

**Returns**:

- A string corresponding to the current value of the servo (no more than 6 characters).

On failure, throws an exception or returns **Y_ADVERTISEDVALUE_INVALID**.
servo→get_enabled()
servo→enabled() servo.get_enabled()

Returns the state of the servos.

int get_enabled()

Returns:

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the servos

On failure, throws an exception or returns Y_ENABLED_INVALID.
servo.get_enabledAtPowerOn()
servo.enabledAtPowerOn()
servo.get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

int get_enabledAtPowerOn()

Returns:
either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the servo signal generator state at power up

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.
YServo

servo → get_errorMessage()
servo → errorMessage() servo.get_errorMessage()

Returns the error message of the latest error with the servo.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
- a string corresponding to the latest error message that occurred while using the servo object
Returns the numerical error code of the latest error with the servo.

```c
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns**:

a number corresponding to the code of the latest error that occurred while using the servo object
servo→get_friendlyName()
servo→friendlyName() servo.get_friendlyName()

Returns a global identifier of the servo in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName( )

The returned string uses the logical names of the module and of the servo if they are defined, otherwise the serial number of the module and the hardware identifier of the servo (for example: MyCustomName.relay1)

Returns :

- a string that uniquely identifies the servo using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
3. Reference

```java
YServo
servo → get_functionDescriptor()

servo → functionDescriptor()

servo.get_functionDescriptor()
```

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```java
String get_functionDescriptor()
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**
- an identifier of type `YFUN_DESCR`.

If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.
servo→get_functionId()
servo→functionId()servo.get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

String get_functionId()

For example relay1

**Returns**:  
a string that identifies the servo (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
servo→get_hardwareId() YServo
servo→hardwareId() servo.get_hardwareId()

Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the servo (for example RELAYL01-123456.relay1).

Returns:
- a string that uniquely identifies the servo (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
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.
servo→get_module()
servo→module()servo.get_module()

Gets the YModule object for the device on which the function is located.

YModule get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns:

an instance of YModule
servo → get_neutral()  
**servo → neutral()**  
**servo.get_neutral()**

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**.
servo\rightarrow \texttt{get\_position()}

\texttt{servo\rightarrow position() \texttt{servo.get\_position()}}

Returns the current servo position.

\begin{verbatim}
int get_position()
\end{verbatim}

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>an integer corresponding to the current servo position</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns \texttt{Y\_POSITION\_INVALID}. 
servo→get_positionAtPowerOn()
servo→positionAtPowerOn()
servo.get_positionAtPowerOn()  

Returns the servo position at device power up.

int get_positionAtPowerOn()

**Returns:**

an integer corresponding to the servo position at device power up

On failure, throws an exception or returns Y_POSITIONATPOWERON_INVALID.
<table>
<thead>
<tr>
<th>YServo</th>
<th>servo→get_range()</th>
<th>servo→range()</th>
<th>servo.get_range()</th>
</tr>
</thead>
</table>

Returns the current range of use of the servo.

```c
int get_range()
```

**Returns:**
- an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns Y_RANGE_INVALID.
servo→getUserData()
servo→userData()\nservo.getUserData()

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.
servo→isOnline() servo.isOnline()  

Checks if the servo is currently reachable, without raising any error.

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

**Returns:**

true if the servo can be reached, and false otherwise
**servo.load()**

Preloads the servo cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>msValidity</strong></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.
servo→move(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.
**YServo nextServo()**

Continues the enumeration of servos started using `yFirstServo()`.

**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.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
### Reference

**YServo**

#### servo->set_enabled()

**servo->setEnabled()**

**servo.set_enabled()**

Stops or starts the servo.

```c
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

```c
YServo

servo.set_enabledAtPowerOn()

Configure the servo signal generator state at power up.

```int set_enabledAtPowerOn( int newval)

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters:**

- `newval` either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Changes the logical name of the servo.

```java
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.

**Parameters:**
- `newval` an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

**Returns:**
- YAPI_SUCCESS if the call succeeds.
- On failure, throws an exception or returns a negative error code.
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.
Configure the servo position at device power up.

```c
int set_positionAtPowerOn( int newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**
- `newval` an integer

**Returns :**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
3. Reference

**servo.set_range()**

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.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>newval</strong> an integer corresponding to the range of use of the servo, specified in per cents</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YAPI_SUCCESS</strong> 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.

```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
3.42. 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 YTtemperature = yoctolib.YTemperature;
require_once('yocto_temperature.php');
```

```php
require_once('yocto_temperature.php');
```

```cpp
#include "yocto_temperature.h"
```

```m
#include "yocto_temperature.h"
```

```pas
uses yocto_temperature;
```

```vb
yocto_temperature.vb
```

```cs
yocto_temperature.cs
```

```java
import com.yoctopuce.YoctoAPI.YTemperature;
```

```python
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.
3. Reference

Temperature→get_highestValue()
Returns the maximal value observed for the temperature since the device was started.

Temperature→get_logFrequency()
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

Temperature→get_logicalName()
Returns the logical name of the temperature sensor.

Temperature→get_lowestValue()
Returns the minimal value observed for the temperature since the device was started.

Temperature→get_module()
Gets the YModule object for the device on which the function is located.

Temperature→get_module_async(callback, context)
Gets the YModule object for the device on which the function is located (asynchronous version).

Temperature→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

Temperature→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

Temperature→get_resolution()
Returns the resolution of the measured values.

Temperature→get_sensorType()
Returns the temperature sensor type.

Temperature→get_unit()
Returns the measuring unit for the temperature.

Temperature→get_userData()
Returns the value of the userData attribute, as previously stored using method set_userData.

Temperature→isOnline()
Checks if the temperature sensor is currently reachable, without raising any error.

Temperature→isOnline_async(callback, context)
Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

Temperature→load(msValidity)
Preloads the temperature sensor cache with a specified validity duration.

Temperature→loadCalibrationPoints(rawValues, refValues)
Retrieves error correction data points previously entered using the method calibrateFromPoints.

Temperature→load_async(msValidity, callback, context)
Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

Temperature→nextTemperature()
Continues the enumeration of temperature sensors started using yFirstTemperature().

Temperature→registerTimedReportCallback(callback)
Registers the callback function that is invoked on every periodic timed notification.

Temperature→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

Temperature→set_highestValue(newval)
Changes the recorded maximal value observed.

Temperature→set_logFrequency(newval)
Changes the datalogger recording frequency for this function.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temperature.set_logicalName(newval)</code></td>
<td>Changes the logical name of the temperature sensor.</td>
</tr>
<tr>
<td><code>temperature.set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>temperature.set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>temperature.set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>temperature.set_sensorType(newval)</code></td>
<td>Modify the temperature sensor type.</td>
</tr>
<tr>
<td><code>temperature.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.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 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()

Starts the enumeration of temperature sensors currently accessible.

YTemperature.FirstTemperature()

Use the method YTemperature.nextTemperature() to iterate on next temperature sensors.

Returns:

A pointer to a YTemperature object, corresponding to the first temperature sensor currently online, or a null pointer if there are none.
3. Reference

YTemperature

**temperature.calibrateFromPoints()**

**Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.**

```
int calibrateFromPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**

- **rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- **refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**

- **YAPI_SUCCESS** if the call succeeds.
- On failure, throws an exception or returns a negative error code.
3. Reference

**YTemperature**

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**String describe( )**

More precisely, **TYPE** is the type of the function, **NAME** is the name used for the first access to the function, **SERIAL** is the serial number of the module if the module is connected or "unresolved", and **FUNCTIONID** is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(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`)
YTemperature

temperature → get_advertisedValue()
temperature → advertisedValue()
temperature.get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

String get_advertisedValue() Returns:

- a string corresponding to the current value of the temperature sensor (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
YTemperature

\[ \text{temperature} \rightarrow \text{get\_currentRawValue()} \]
\[ \text{temperature} \rightarrow \text{currentRawValue()} \]
\[ \text{temperature.get\_currentRawValue()} \]

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}. 
| temperature → get_currentValue() | YTemperature |
| temperature → currentValue() |
| temperature.get_currentValue() |

Returns the current value of the temperature, in Celsius, as a floating point number.

```java
double get_currentValue()
```

**Returns:**
- a floating point number corresponding to the current value of the temperature, in Celsius, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`. 
temperature → get_errorMessage()
temperature → errorMessage()
temperature.get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
  a string corresponding to the latest error message that occurred while using the temperature sensor object
Returns the numerical error code of the latest error with the temperature sensor.

**int get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

- a number corresponding to the code of the latest error that occurred while using the temperature sensor object
temperature→get_friendlyName()

temperature→friendlyName()

temperature.get_friendlyName()

**YTemperature**

**Returns a global identifier of the temperature sensor in the format**

`MODULE_NAME.FUNCTION_NAME`.

**String get_friendlyName()**

The returned string uses the logical names of the module and of the temperature sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the temperature sensor (for example: `MyCustomName.relay1`)

**Returns:**

A string that uniquely identifies the temperature sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
3. Reference

**String `get_functionDescriptor()`**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>an identifier of type <code>YFUN_DESCR</code>.</td>
</tr>
</tbody>
</table>

**If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTION_INVALID`**.
`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`. 

---

**YTemperature**

`temperature.functionId()`
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.
YTemperature

temperature→get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

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.
YTemperature

**temperature.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**String get_logFrequency()**

**Returns:**
- a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns **Y_LOGFREQUENCY_INVALID**.
YTemperature

temperature.get_logicalName()

Returns the logical name of the temperature sensor.

String get_logicalName()

Returns:

- a string corresponding to the logical name of the temperature sensor.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
Returns the minimal value observed for the temperature since the device was started.

Returns:
- a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.
3. Reference

**YTemperature**

`temperature.get_module()`

`temperature.module()`

`temperature.get_module()`

Gets the `YModule` object for the device on which the function is located.

`YModule get_module()`

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns:**

- an instance of `YModule`
YTemperature

get_recordedData()

recordedData()

temperature.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet get_recordedData(long startTime, long endTime)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startTime</strong> 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> 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>

<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>
YTemperature

temperature.get_reportFrequency()  returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String get_reportFrequency() ( )

Returns:

- a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.


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}.
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`. 
3. Reference

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`. 
YTemperature

temperature→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

Object get_userData( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns:
the object stored previously by the caller.
**YTemperature**

3. Reference

<table>
<thead>
<tr>
<th>temperature</th>
<th>→ isOnline()</th>
<th>temperature.isOnline()</th>
</tr>
</thead>
</table>

Checks if the temperature sensor is currently reachable, without raising any error.

```java
boolean isOnline()
```

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

**Returns:**

- true if the temperature sensor can be reached, and false otherwise
### Temperature

**temperature→load()**

Preloads the temperature sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- **msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
### temperature.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
YTemperature

**nextTemperature()**

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.
YTemperature

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.
**temperature.registerValueCallback()**  

`temperature.registerValueCallback()` Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
3. Reference

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.

```java
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.

```java
int set_logicalName(String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters:**
- `newval` a string corresponding to the logical name of the temperature sensor.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
temperature→set_lowestValue()

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.
YTemperature

temperature→set_reportFrequency()
temperature→setReportFrequency()
temperature.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

**temperature → set_sensorType()**

**temperature → setSensorType()**

**temperature.set_sensorType()**

Modify the temperature sensor type.

**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 the module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters:**

- **newval** a value among `Y_SENSORTYPE_DIGITAL`, `Y_SENSORTYPE_TYPE_K`, `Y_SENSORTYPE_TYPE_E`, `Y_SENSORTYPE_TYPE_J`, `Y_SENSORTYPE_TYPE_N`, `Y_SENSORTYPE_TYPE_R`, `Y_SENSORTYPE_TYPE_S`, `Y_SENSORTYPE_TYPE_T`, `Y_SENSORTYPE_PT100_4WIRES`, `Y_SENSORTYPE_PT100_3WIRES` and `Y_SENSORTYPE_PT100_2WIRES`

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
temperature->set_userData()

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
3.43. 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>
```

```
node
var yoctolib = require('yoctolib');
var YTilt = yoctolib.YTilt;
```

```
php
require_once('yocto_tilt.php');
```

```
cpp
#include "yocto_tilt.h"
```

```
m
#include "yocto_tilt.h"
```

```
pas
uses yocto_tilt;
```

```
vb
yocto_tilt.vb
```

```
java
import com.yoctopuce.YoctoAPI.YTilt;
```

```
py
from yocto_tilt import *
```

### Global functions

**yFindTilt(func)**

Retrieves a tilt sensor for a given identifier.

**yFirstTilt()**

Starts the enumeration of tilt sensors currently accessible.

### YTilt methods

**tilt→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

**tilt→describe()**

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**tilt→get_advertisedValue()**

Returns the current value of the tilt sensor (no more than 6 characters).

**tilt→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, 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`. 
<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 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>
</tbody>
</table>
### Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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()

`YTilt.FindTilt(String func)`

Retrieves a tilt 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 tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters:**

- `func` a string that uniquely characterizes the tilt sensor

**Returns:**

- a `YTilt` object allowing you to drive the tilt sensor.
YTilt.FirstTilt()
YTilt.FirstTilt()

Starts the enumeration of tilt sensors currently accessible.

Y\texttt{Tilt FirstTilt()}

Use the method \texttt{YTilt.nextTilt()} to iterate on next tilt sensors.

\textbf{Returns :}

a pointer to a \texttt{YTilt} object, corresponding to the first tilt sensor currently online, or a null pointer if there are none.
ytilt→calibrateFromPoints()

tilt.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints(ArrayList<Double> rawValues,
                        ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Returns a short text that describes unambiguously the instance of the tilt sensor in the form \( \text{TYPE}(\text{NAME})=\text{SERIAL}.\text{FUNCTIONID} \).

\begin{verbatim}
String describe()

More precisely, \text{TYPE} is the type of the function, \text{NAME} its 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 tilt sensor (ex: \text{Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1})
\end{verbatim}
Returns the current value of the tilt sensor (no more than 6 characters).

String `get_advertisedValue()`

**Returns:**
- a string corresponding to the current value of the tilt sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
3. Reference

**YTilt**

tilt→get_currentRawValue()
tilt→currentRawValue()
tilt.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, 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, as a floating point number

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
Returns the current value of the inclination, in degrees, as a floating point number.

```java
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`. 
3. Reference

\texttt{tilt\rightarrow get\_errorMessage()}\hspace{1cm} \texttt{YTilt}

\texttt{tilt\rightarrow errorMessage()}\hspace{1cm} \texttt{tilt.get\_errorMessage()}

Returns the error message of the latest error with the tilt sensor.

\texttt{String get\_errorMessage()}\hspace{1cm}

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 tilt sensor object
Ytilt

get_errorType()

tilt.get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

int get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:

a number corresponding to the code of the latest error that occurred while using the tilt sensor object
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_friendlyName()</code></td>
<td>Returns a global identifier of the tilt sensor in the format <code>MODULE_NAME.FUNCTION_NAME</code>.</td>
</tr>
<tr>
<td><code>friendlyName()</code></td>
<td>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: <code>MyCustomName.relay1</code>).</td>
</tr>
</tbody>
</table>

**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`. 


štít→\texttt{get\_functionDescriptor}()
štít→\texttt{functionDescriptor}()
štít.\texttt{get\_functionDescriptor}()

Returns a unique identifier of type \texttt{YFUN\_DESCR} corresponding to the function.

\texttt{String 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.

\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}.}
3. Reference

<table>
<thead>
<tr>
<th>tilt→get_functionId()</th>
<th>YTilt</th>
</tr>
</thead>
<tbody>
<tr>
<td>tilt→functionId()tilt.get_functionId()</td>
<td></td>
</tr>
</tbody>
</table>

Returns the hardware identifier of the tilt sensor, without reference to the module.

```java
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.
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String \texttt{get\_logFrequency()} 

\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}

On failure, throws an exception or returns \texttt{Y\_LOGFREQUENCY\_INVALID}. 
Returns the logical name of the tilt sensor.

<table>
<thead>
<tr>
<th>String get_logicalName()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Returns:</strong></td>
</tr>
<tr>
<td>a string corresponding to the logical name of the tilt sensor.</td>
</tr>
<tr>
<td>On failure, throws an exception or returns Y_LOGICALNAME_INVALID.</td>
</tr>
</tbody>
</table>
Returns the minimal value observed for the inclination since the device was started.

```java
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`. 

---

### Reference

YTilt

```java
tilt.get_lowestValue()
```

`tilt.get_lowestValue()`

Returns the minimal value observed for the inclination since the device was started.
3. Reference

tilt→get_module() YTilt
tilt→module() tilt.get_module()  

Gets the YModule object for the device on which the function is located.

YModule get_module()  

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns:
- an instance of YModule
tilt→get_recordedData()
	return recordedData()

YDataSet get_recordedData( long startTime, long endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

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

Ytilt

tilt\to get_reportFrequency()
tilt\to reportFrequency()
tilt.\textit{get_reportFrequency}()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String \textit{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.

```
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 inclination.

```java
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.
Checks if the tilt sensor is currently reachable, without raising any error.

**boolean isOnline()**

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

**Returns:**
- true if the tilt sensor can be reached, and false otherwise
Preloads the tilt sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

### Parameters :
- **msValidity**
  - an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

### Returns :
- **YAPI_SUCCESS** when the call succeeds.

On failure, throws an exception or returns a negative error code.
### Tilt\rightarrow\text{loadCalibrationPoints}() \quad \text{YTilt}

tilt.loadCalibrationPoints()

Receives error correction data points previously entered using the method \text{calibrateFromPoints}.

```java
int loadCalibrationPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- **rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

- **refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- \text{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 `yFirstTilt()`.

**YTilt nextTilt()**

**Returns:**

A pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a null pointer if there are no more tilt sensors to enumerate.
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.
tilt.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
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.
tilt->set_logFrequency()
tilt->setLogFrequency() tilt.set_logFrequency()

Changes the datalogger recording frequency for this function.

```plaintext
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.

```java
int set_logicalName(String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters:**
- `newval` a string corresponding to the logical name of the tilt sensor.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
### tilt.set_lowestValue()

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.
tilt.set_reportFrequency()

Changes the timed value notification frequency for this function.

```java
int set_reportFrequency(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 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.

```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
3.44. 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"
```
```
pas
uses yocto_voc;
```
```
vb
yocto_voc.vb
```
```
java
import com.yoctopuce.YoctoAPI.YVoc;
```
```
py
from yocto_voc import *
```

**Global functions**

yFindVoc(func)

Retrieves a Volatile Organic Compound sensor for a given identifier.

yFirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

**YVoc methods**

voc → calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voc → describe()

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form TYPE (NAME) = SERIAL.FUNCTIONID.

voc → get_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

voc → get_currentRawValue()

Returns the 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 MODULE_NAME.FUNCTION_NAME.

voc → get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

voc → get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

voc → get_hardwareId()
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>voc-&gt;get_highestValue()</code></td>
<td>Returns the maximal value observed for the estimated VOC concentration since the device was started.</td>
</tr>
<tr>
<td><code>voc-&gt;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>voc-&gt;get_logicalName()</code></td>
<td>Returns the logical name of the Volatile Organic Compound sensor.</td>
</tr>
<tr>
<td><code>voc-&gt;get_lowestValue()</code></td>
<td>Returns the minimal value observed for the estimated VOC concentration since the device was started.</td>
</tr>
<tr>
<td><code>voc-&gt;get_module()</code></td>
<td>Gets the YModule object for the device on which the function is located.</td>
</tr>
<tr>
<td><code>voc-&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>voc-&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>voc-&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>voc-&gt;get_resolution()</code></td>
<td>Returns the resolution of the measured values.</td>
</tr>
<tr>
<td><code>voc-&gt;get_unit()</code></td>
<td>Returns the measuring unit for the estimated VOC concentration.</td>
</tr>
<tr>
<td><code>voc-&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>voc-&gt;isOnline()</code></td>
<td>Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><code>voc-&gt;isOnline_async(callback, context)</code></td>
<td>Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><code>voc-&gt;load(msValidity)</code></td>
<td>Preloads the Volatile Organic Compound sensor cache with a specified validity duration.</td>
</tr>
<tr>
<td><code>voc-&gt;loadCalibrationPoints(rawValues, refValues)</code></td>
<td>Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code>.</td>
</tr>
<tr>
<td><code>voc-&gt;load_async(msValidity, callback, context)</code></td>
<td>Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><code>voc-&gt;nextVoc()</code></td>
<td>Continues the enumeration of Volatile Organic Compound sensors started using <code>yFirstVoc()</code>.</td>
</tr>
<tr>
<td><code>voc-&gt;registerTimedReportCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every periodic timed notification.</td>
</tr>
<tr>
<td><code>voc-&gt;registerValueCallback(callback)</code></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
</tbody>
</table>
| `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><code>voc-&gt;set_logFrequency(newval)</code></td>
<td>Changes the datalogger recording frequency for this function.</td>
</tr>
<tr>
<td><code>voc-&gt;set_logicalName(newval)</code></td>
<td>Changes the logical name of the Volatile Organic Compound sensor.</td>
</tr>
<tr>
<td><code>voc-&gt;set_lowestValue(newval)</code></td>
<td>Changes the recorded minimal value observed.</td>
</tr>
<tr>
<td><code>voc-&gt;set_reportFrequency(newval)</code></td>
<td>Changes the timed value notification frequency for this function.</td>
</tr>
<tr>
<td><code>voc-&gt;set_resolution(newval)</code></td>
<td>Changes the resolution of the measured physical values.</td>
</tr>
<tr>
<td><code>voc-&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>voc-&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>
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.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>func</strong> a string that uniquely characterizes the Volatile Organic Compound sensor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a <em>YVoc</em> object allowing you to drive the Volatile Organic Compound sensor.</td>
</tr>
</tbody>
</table>
YVoc.FirstVoc()

Use the method `YVoc.nextVoc()` to iterate on next Volatile Organic Compound sensors.

**Returns:**
- a pointer to a `YVoc` object, corresponding to the first Volatile Organic Compound sensor currently online, or
- a null pointer if there are none.
voc->calibrateFromPoints()

voc.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```java
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 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.
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`. 
Returns the error message of the latest error with the Volatile Organic Compound sensor.

String getErrorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
a string corresponding to the latest error message that occurred while using the Volatile Organic Compound sensor object
voc\texttt{\rightarrow \text{get\_errorType()}}

\texttt{voc\rightarrow \text{errorType()}} \texttt{voc\cdot \text{get\_errorType()}}

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

\begin{verbatim}
int get_errorType()
\end{verbatim}

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

\textbf{Returns}:

\begin{quote}
\begin{itemize}
\item a number corresponding to the code of the latest error that occurred while using the Volatile Organic Compound sensor object
\end{itemize}
\end{quote}
3. Reference

```
voc->get_friendlyName()
```

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```
String get_friendlyName()
```

The returned string uses the logical names of the module and of the Volatile Organic Compound sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the Volatile Organic Compound sensor (for example: `MyCustomName.relay1`)

**Returns:**
- a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`. 
voc → get_functionDescriptor() YVoc
voc → functionDescriptor()
voc.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
YVoc

voc→get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

String get_functionId()

For example relay1

**Returns:**

a string that identifies the Volatile Organic Compound sensor (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
voc → get_hardwareId()

YVoc

voc → hardwareId()
voc.get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form
SERIALFUNCTIONID.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier
of the Volatile Organic Compound sensor (for example RELAYLO1-123456.relay1).

Returns:

- a string that uniquely identifies the Volatile Organic Compound sensor (ex: RELAYLO1–123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
### 3. Reference

**voc.get_highestValue()**

Returns the maximal value observed for the estimated VOC concentration since the device was started.

```java
double get_highestValue()
```

**Returns:**

A floating point number corresponding to the maximal value observed for the estimated VOC concentration since the device was started.

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`. 
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String get_logFrequency() {  

Returns:

- a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.
voc→get_logicalName()
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.
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()
voc→module()voc.get_module()

Gets the YModule object for the device on which the function is located.

YModule get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns:
an instance of YModule
3. Reference

**YVoc**

voc → get_recordedData()

voc → recordedData()

voc.get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

| YDataSet get_recordedData( long startTime, long endTime) |

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters:**

- `startTime` the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any 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

YVoc

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.
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→getUserData()

`voc.getUserData()`

YVoc

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.
voc→isOnline() \( \text{voc}.\text{isOnline}() \)

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

\[
\text{boolean } \text{isOnline}()\]

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
Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
YVoc

voc.loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```java
int loadCalibrationPoints(ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**

- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
YVoc \rightarrow \text{nextVoc}() \text{ YVoc}

Continues the enumeration of Volatile Organic Compound sensors started using \text{yFirstVoc}().

YVoc \text{ nextVoc}()

\textbf{Returns :}

- a pointer to a YVoc object, corresponding to a Volatile Organic Compound sensor currently online, or a null pointer if there are no more Volatile Organic Compound sensors to enumerate.
voc→registerTimedReportCallback()

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.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
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 Volatile Organic Compound sensor.

```java
int set_logicalName( String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters:**
- `newval` a string corresponding to the logical name of the Volatile Organic Compound sensor.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
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.

```cpp
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.

```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. Reference

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:

```
js
<script type='text/javascript' src='yocto_voltage.js'></script>

nodejs
var yoctolib = require('yoctolib');
var YVoltage = yoctolib.YVoltage;

php
require_once('yocto_voltage.php');

cpp
#include "yocto_voltage.h"

m
#include "yocto_voltage.h"

pas
uses yocto_voltage;

vb
yocto_voltage.vb

cs
yocto_voltage.cs

java
import com.yoctopuce.YoctoAPI.YVoltage;

py
from yocto_voltage import *
```

<table>
<thead>
<tr>
<th>Global functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>yFindVoltage(func)</td>
<td>Retrieves a voltage sensor for a given identifier.</td>
</tr>
<tr>
<td>yFirstVoltage()</td>
<td>Starts the enumeration of voltage sensors currently accessible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YVoltage methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>voltage-&gt;calibrateFromPoints(rawValues, refValues)</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>voltage-&gt;describe()</td>
<td>Returns a short text that describes unambiguously the instance of the voltage sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td>voltage-&gt;get_advertisedValue()</td>
<td>Returns the current value of the voltage sensor (no more than 6 characters).</td>
</tr>
<tr>
<td>voltage-&gt;get_currentRawValue()</td>
<td>Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.</td>
</tr>
<tr>
<td>voltage-&gt;get_currentValue()</td>
<td>Returns the current value of the voltage, in Volt, as a floating point number.</td>
</tr>
<tr>
<td>voltage-&gt;get_errorMessage()</td>
<td>Returns the error message of the latest error with the voltage sensor.</td>
</tr>
<tr>
<td>voltage-&gt;get_errorType()</td>
<td>Returns the numerical error code of the latest error with the voltage sensor.</td>
</tr>
<tr>
<td>voltage-&gt;get_friendlyName()</td>
<td>Returns a global identifier of the voltage sensor in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td>voltage-&gt;get_functionDescriptor()</td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td>voltage-&gt;get_functionId()</td>
<td>Returns the hardware identifier of the voltage sensor, without reference to the module.</td>
</tr>
<tr>
<td>voltage-&gt;get_hardwareId()</td>
<td>Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.</td>
</tr>
</tbody>
</table>
3. Reference

- **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.
3. Reference

<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.

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.
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

voltage→calibrateFromPoints()

voltage.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```java
int calibrateFromPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters:**
- `rawValues` array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
- `refValues` array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
voltage.describe()\voltage.describe()

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 it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:

a string that describes the voltage sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
voltage→get_advertisedValue()
voltage→advertisedValue()
voltage.get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

String get_advertisedValue()

Returns:

a string corresponding to the current value of the voltage sensor (no more than 6 characters).

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

Returns:
- a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.
voltage\(\rightarrow\)get\_current\_Value() 
voltage\(\rightarrow\)current\_Value() 
voltage\.get\_current\_Value()

Returns the current value of the voltage, in Volt, as a floating point number.

\[
\text{double get\_current\_Value( )}
\]

**Returns**:

a floating point number corresponding to the current value of the voltage, in Volt, as a floating point number

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.
voltage → get_errorMessage()
voltage → errorMessage()
voltage.get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

- a string corresponding to the latest error message that occurred while using the voltage sensor object
voltage→get_errorType()

voltage→get_errorType()\voltage.get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

int get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

a number corresponding to the code of the latest error that occurred while using the voltage sensor object
Returns a global identifier of the voltage sensor in the format \texttt{MODULE\_NAME.FUNCTION\_NAME}.

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: \texttt{MyCustomName.relay1})

\textbf{Returns :}

a string that uniquely identifies the voltage sensor using logical names (ex: \texttt{MyCustomName.relay1})

On failure, throws an exception or returns \texttt{Y\_FRIENDLYNAME\_INVALID}.
voltage→get_functionDescriptor()
voltage→functionDescriptor()
voltage.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

String get_functionDescriptor()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns:
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
3. Reference

\texttt{voltage \rightarrow get\_functionId()} \hspace{1cm} \texttt{YVoltage}

\texttt{voltage \rightarrow functionId()} \texttt{voltage.get\_functionId()}

Returns the hardware identifier of the voltage sensor, without reference to the module.

\textbf{String get\_functionId( )}

For example \texttt{relay1}

\begin{itemize}
  \item \textbf{Returns :}
    \begin{itemize}
      \item a string that identifies the voltage sensor (ex: \texttt{relay1})
    \end{itemize}
  \end{itemize}

On failure, throws an exception or returns \texttt{Y\_FUNCTIONID\_INVALID}. 
YVoltage
voltage→get_hardwareId()
voltage→hardwareId()voltage.get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor (for example RELAYLO1-123456.relay1).

Returns:

- a string that uniquely identifies the voltage sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
Returns the maximal value observed for the voltage since the device was started.

\[
\text{double } \textit{get\_highestValue()} \
\]

Returns:
- a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns \textit{Y\_HIGHESTVALUE\_INVALID}.
voltage \rightarrow \text{get\_logFrequency()}

voltage \rightarrow \text{logFrequency()}

voltage.get\_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

String \text{get\_logFrequency()}()

**Returns:**

- a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.
voltage→get_logicalName()
voltage.logicalName()
voltage.get_logicalName()

Returns the logical name of the voltage sensor.

String get_logicalName()

**Returns:**

a string corresponding to the logical name of the voltage sensor.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
Returns the minimal value observed for the voltage since the device was started.

```java
double get_lowestValue() {
    // implementation
    return 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`. 
`YVoltage` 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

```java
voltage→get_recordedData()
voltage→recordedData()
voltage.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.
3. Reference

YVoltage

voltage.get_reportFrequency()
voltage.reportFrequency()
voltage.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

String get_reportFrequency()

Returns:
- A string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.
YVoltage

voltage.get_resolution()

voltage.resolution()voltage.get_resolution()

Returns the resolution of the measured values.

double get_resolution()

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns:
- a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.
3. Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>voltage.get_unit()</code></td>
<td>Returns the measuring unit for the voltage.</td>
</tr>
</tbody>
</table>
| `String get_unit()`   | Returns:
|                       | a string corresponding to the measuring unit for the voltage |
|                       | On failure, throws an exception or returns `Y_UNIT_INVALID`. |
voltage→get_userData()
voltage→userData()voltage.get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

Object get_userData( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns:
the object stored previously by the caller.
voltage → isOnline()\texttt{voltage.isOnline()}

Checks if the voltage sensor is currently reachable, without raising any error.

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:**
- \texttt{true} if the voltage sensor can be reached, and \texttt{false} otherwise
**voltage.load()**

Preloads the voltage sensor cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
voltage→loadCalibrationPoints()

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```java
int loadCalibrationPoints( ArrayList<Double> rawValues, ArrayList<Double> refValues)
```

**Parameters:**
- `rawValues` array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- `refValues` array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
**voltage → nextVoltage()**  
*YVoltage*  

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

### YVoltage nextVoltage()  

**Returns:**  
a pointer to a *YVoltage* object, corresponding to a voltage sensor currently online, or a *null* pointer if there are no more voltage sensors to enumerate.
YVoltage → voltage.registerTimedReportCallback()

voltage.registerTimedReportCallback()

Registers the callback function that is invoked on every periodic timed notification.

`int registerTimedReportCallback( TimedReportCallback callback)`

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.
voltage→registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of ySleep or yHandleEvents. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- **callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
voltage→set_highestValue()  
voltage→setHighestValue()  
voltage.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 voltage sensor.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newval</td>
<td>a string corresponding to the logical name of the voltage sensor.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAPI_SUCCESS</td>
<td>if the call succeeds.</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns a negative error code.
voltage→set_lowestValue()
voltage→setLowestValue()
voltage.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.
voltage.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.

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.

```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
3.46. 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:

```javascript
<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 <code>TYPE(NAME)=SERIAL.FUNCTIONID</code>.</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 <code>MODULE_NAME.FUNCTION_NAME</code>.</td>
</tr>
<tr>
<td><code>vsource→get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type <code>YFUN_DESCR</code> 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 <code>SERIAL.FUNCTIONID</code>.</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>
</tbody>
</table>
| `vsource→get_module_async(callback, context)` | }
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vsource</strong>→get_overCurrent()</td>
<td>Gets the YModule object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_overCurrent()</td>
<td>Returns true if the appliance connected to the device is too greedy.</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_overHeat()</td>
<td>Returns true if the module is overheating.</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_overLoad()</td>
<td>Returns true if the device is not able to maintain the requested voltage output.</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_regulationFailure()</td>
<td>Returns true if the voltage output is too high regarding the requested voltage.</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_unit()</td>
<td>Returns the measuring unit for the voltage.</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_userData()</td>
<td>Returns the value of the userData attribute, as previously stored using method set_userData.</td>
</tr>
<tr>
<td><strong>vsource</strong>→get_voltage()</td>
<td>Returns the voltage output command (mV)</td>
</tr>
<tr>
<td><strong>vsource</strong>→isOnline()</td>
<td>Checks if the function is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><strong>vsource</strong>→isOnline_async(callback, context)</td>
<td>Checks if the function is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><strong>vsource</strong>→load(msValidity)</td>
<td>Preloads the function cache with a specified validity duration.</td>
</tr>
<tr>
<td><strong>vsource</strong>→load_async(msValidity, callback, context)</td>
<td>Preloads the function cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><strong>vsource</strong>→nextVSource()</td>
<td>Continues the enumeration of voltage sources started using yFirstVSource().</td>
</tr>
<tr>
<td><strong>vsource</strong>→pulse(voltage, ms_duration)</td>
<td>Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.</td>
</tr>
<tr>
<td><strong>vsource</strong>→registerValueCallback(callback)</td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td><strong>vsource</strong>→set_logicalName(newval)</td>
<td>Changes the logical name of the voltage source.</td>
</tr>
<tr>
<td><strong>vsource</strong>→set_userData(data)</td>
<td>Stores a user context provided as argument in the userData attribute of the function.</td>
</tr>
<tr>
<td><strong>vsource</strong>→set_voltage(newval)</td>
<td>Tunes the device output voltage (milliVolts).</td>
</tr>
<tr>
<td><strong>vsource</strong>→voltageMove(target, ms_duration)</td>
<td>Performs a smooth move at constant speed toward a given value.</td>
</tr>
<tr>
<td><strong>vsource</strong>→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

**YVSource**

**YVSource.FindVSource()**

Retrieves a voltage source for a given identifier.

```plaintext
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()
YVSource.FirstVSource()

Starts the enumeration of voltage sources currently accessible.

YVSource FirstVSource()

yFirstVSource() — YVSource.FirstVSource() YVSource.FirstVSource()

Starts the enumeration of voltage sources currently accessible.

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 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 function (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
Returns the current value of the voltage source (no more than 6 characters).

```
String get_advertisedValue()
```

Returns:
- a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.
Returns the error message of the latest error with this function.

This method is mostly useful when using the Yoctopuce library with exceptions disabled.
Returns the numerical error code of the latest error with this function.

```vbnet
vsource.get_errorType()
```

```javascript
function get_errorType() {
}
```

```php
function get_errorType() {
}
```

```cpp
YRETCODE get_errorType() {
}
```

```pas
function get_errorType: YRETCODE {
}
```

```java
int get_errorType() {
}
```

```python
def get_errorType() {
}
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

A number corresponding to the code of the latest error that occurred while using this function object.
Returns true if external power supply voltage is too low.

```
int get_extPowerFailure()
```

Returns : 
either Y_EXTPOWERFAILURE_FALSE or Y_EXTPOWERFAILURE_TRUE, according to true if external power supply voltage is too low

On failure, throws an exception or returns Y_EXTPOWERFAILURE_INVALID.
vsourse→get_failure() (vsourse.get_failure())

Returns true if the module is in failure mode.

int get_failure()

vsourse→get_failure() (vsourse.get_failure())

Returns true if the module is in failure mode.

- js: function get_failure()
- php: function get_failure()
- cpp: Y_FAILURE_enum get_failure()
- m: -(Y_FAILURE_enum) failure
- pas: function get_failure(): Integer
- vb: function get_failure() As Integer
- cs: int get_failure()
- java: int get_failure()
- py: def get_failure()
- cmd: YVSource target get_failure

More information can be obtained by testing get_overheat, get_overcurrent etc... When an error condition is met, the output voltage is set to zero and cannot be changed until the reset() function is called.

Returns:

either Y_FAILURE_FALSE or Y_FAILURE_TRUE, according to true if the module is in failure mode.

On failure, throws an exception or returns Y_FAILURE_INVALID.
vsource→get_friendlyName()
vsource→friendlyName()
vsource.get_friendlyName()

Returns a global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

String get_friendlyName()  
The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

Returns:
   a string that uniquely identifies the function using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
Returns a unique identifier of type YFUN_DESCR corresponding to the function.

Returns : 

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.
vsource→get_functionId()

vsource→functionId() vsource.get_vsourceId()

Returns the hardware identifier of the function, without reference to the module.

String get_functionId()

For example relay1

Returns:

A string that identifies the function (ex: relay1) On failure, throws an exception or returns Y_FUNCTIONID_INVALID.
Returns the unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

String `get_hardwareId()`
The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function. (for example RELAYLO1-123456.relay1)

Returns:
- a string that uniquely identifies the function (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`. 
vsource→get_logicalName()
vsource→logicalName()
vsource.get_logicalName()

Returns the logical name of the voltage source.

String get_logicalName()

vsource→get_logicalName()
vsource→logicalName()vsource.get_logicalName()

Returns the logical name of the voltage source.

| js    | function get_logicalName() |
| php   | function get_logicalName() |
| cpp   | string get_logicalName()   |
| m     | -(NSString*) logicalName   |
| pas   | function get_logicalName() : string |
| vb    | function get_logicalName() As String |
| ca    | string get_logicalName()   |
| java  | String get_logicalName()   |
| py    | def get_logicalName()      |
| cmd   | YVSource target get_logicalName |

Returns:

a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.
YModule get_module()

Gets the YModule object for the device on which the function is located.

Returns:

an instance of YModule
Vsource\rightarrow get\_overCurrent() \vspace{2px}
Vsource\rightarrow overCurrent() \vspace{2px}
Vsource.get\_overCurrent() \vspace{2px}

Returns true if the appliance connected to the device is too greedy.

int get\_overCurrent() \vspace{2px}

Vsource\rightarrow get\_overCurrent() \vspace{2px}
Vsource\rightarrow overCurrent()Vsource.get\_overCurrent() \vspace{2px}

Returns true if the appliance connected to the device is too greedy.

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.
YVSource

YVSource::get_overHeat() Returns TRUE if the module is overheating.

int get_overHeat()

YVSource::get_overHeat() Returns TRUE if the module is overheating.

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.
Returns true if the device is not able to maintain the requested voltage output.

```plaintext
int 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`.
3. Reference

YVSource

target get_regulationFailure

Returns true if the voltage output is too high regarding the requested voltage.

int get_regulationFailure()

YVSource

target get_regulationFailure

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

Returns the measuring unit for the voltage.

String `get_unit()`

YVSource

Returns the measuring unit for the voltage.

<table>
<thead>
<tr>
<th>js</th>
<th>function <code>get_unit()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>php</td>
<td>function <code>get_unit()</code></td>
</tr>
<tr>
<td>cpp</td>
<td>string <code>get_unit()</code></td>
</tr>
<tr>
<td>m</td>
<td>-(NSString*) <code>unit</code></td>
</tr>
<tr>
<td>pas</td>
<td>function <code>get_unit()</code>: string</td>
</tr>
<tr>
<td>vb</td>
<td>function <code>get_unit()</code> As String</td>
</tr>
<tr>
<td>cs</td>
<td>string <code>get_unit()</code></td>
</tr>
<tr>
<td>java</td>
<td>String <code>get_unit()</code></td>
</tr>
<tr>
<td>py</td>
<td>def <code>get_unit()</code></td>
</tr>
<tr>
<td>cmd</td>
<td>YVSource target <code>get_unit</code></td>
</tr>
</tbody>
</table>

**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.
3. Reference

**YVSource**

\[ \text{vsource} \rightarrow \text{get\_voltage() \ vsource} \rightarrow \text{voltage()} \ vsource \rightarrow \text{get\_voltage() vsource.get\_voltage() } \]

Returns the voltage output command (mV)

\[ \text{int get\_voltage( )} \]

**vsource \rightarrow get\_voltage() \ vsource \rightarrow voltage() \ vsource \rightarrow get\_voltage() \ vsource.get\_voltage()**

Returns the voltage output command (mV)

\[ \begin{array}{l}
\text{js} \quad \text{function get\_voltage( )} \\
\text{php} \quad \text{function get\_voltage( )} \\
\text{cpp} \quad \text{int get\_voltage( )} \\
\text{m} \quad -(\text{int}) \text{voltage} \\
\text{pas} \quad \text{function get\_voltage( ): LongInt} \\
\text{vb} \quad \text{function get\_voltage( ) As Integer} \\
\text{cs} \quad \text{int get\_voltage( )} \\
\text{java} \quad \text{int get\_voltage( )} \\
\text{py} \quad \text{def get\_voltage( )} \\
\end{array} \]

**Returns**:

an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns Y_VOLTAGE_INVALID.
Checks if the function is currently reachable, without raising any error.

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.

**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()`.

```javascript
function nextVSource() {
  // Implementation...
}
```

```php
function nextVSource() {
  // Implementation...
}
```

```cpp
YVSource * nextVSource() {
  // Implementation...
}
```

```m
-(YVSource*) nextVSource
```

```pas
function nextVSource( ): TYVSource
```

```vb
function nextVSource( ) As YVSource
```

```cs
YVSource nextVSource( )
```

```java
YVSource nextVSource( )
```

```py
def nextVSource( )
```

**Returns:**
- A pointer to a `YVSource` object, corresponding to a voltage source currently online, or a null pointer if there are no more voltage sources to enumerate.
Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.

```python
int pulse(int voltage, int ms_duration)
```

**Parameters:**

- `voltage`: pulse voltage, in millivolts
- `ms_duration`: pulse duration, in milliseconds

**Returns:**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
vsource→registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

void registerValueCallback( UpdateCallback callback)

The callback is invoked only during the execution of ySleep or yHandleEvents. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters:

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
changes the logical name of the voltage source.

```cpp
int set_logicalName( String newval)
```

```java
int set_logicalName( String newval)
```

```js
function set_logicalName( newval)
```

```php
function set_logicalName( $newval)
```

```cpp
int set_logicalName( const string& newval)
```

```m
-(int) setLogicalName : (NSString*) newval
```

```pas
function set_logicalName( newval: string); integer
```

```vb
function set_logicalName( ByVal newval As String) As Integer
```

```cs
int set_logicalName( string newval)
```

```java
int set_logicalName( String newval)
```

```py
def set_logicalName( newval)
```

```cmd
YVSource target set_logicalName newval
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

- `newval` a string corresponding to the logical name of the voltage source

**Returns :**

- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
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)
```

```objective-c
-(void) setUserData : (void*) data
```

```pas```
```
procedure set_userData( data: TObject)
```

```vb```
```
procedure set_userData( ByVal data As Object)
```

```java```
```
void set_userData( Object data)
```

```python```
```
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

YVSource

**vsoure→set_voltage()**

Tunes the device output voltage (milliVolts).

```
int set_voltage( int newval)
```

**vsoure→setVoltage()**

Tunes the device output voltage (milliVolts).

```
set_voltage( newval)
```

**Parameters**:

- **newval** an integer

**Returns**:

- **YAPI_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.
vsource→voltageMove()\texttt{vsource.voltageMove()}

Perform a smooth move at constant speed toward a given value.

\begin{verbatim}
int voltageMove(int target, int ms_duration)
\end{verbatim}

Parameters:
- \texttt{target} new output value at end of transition, in milliVolts.
- \texttt{ms_duration} transition duration, in milliseconds

Returns:
- \texttt{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"
```
```python
#import "yocto_wakeupmonitor.h"
```
```pascal
uses yocto_wakeupmonitor;
```
```vb
yocto_wakeupmonitor.vb
```
```csharp
yocto_wakeupmonitor.cs
```
```java
import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
```
```python
from yocto_wakeupmonitor import *
```

## Global functions

- **yFindWakeUpMonitor**(<em>func</em>)
  Retrieves a monitor for a given identifier.

- **yFirstWakeUpMonitor**()
  Starts the enumeration of monitors currently accessible.

## YWakeUpMonitor methods

- **wakeupmonitor→describe()**
  Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **wakeupmonitor→get_advertisedValue()**
  Returns the current value of the monitor (no more than 6 characters).

- **wakeupmonitor→get_errorMessage()**
  Returns the error message of the latest error with the monitor.

- **wakeupmonitor→get_errorType()**
  Returns the numerical error code of the latest error with the monitor.

- **wakeupmonitor→get_friendlyName()**
  Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

- **wakeupmonitor→get_functionDescriptor()**
  Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **wakeupmonitor→get_functionId()**
  Returns the hardware identifier of the monitor, without reference to the module.

- **wakeupmonitor→get_hardwareId()**
  Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.

- **wakeupmonitor→get_logicalName()**
  Returns the logical name of the monitor.

- **wakeupmonitor→get_module()**
  Gets the `YModule` object for the device on which the function is located.

- **wakeupmonitor→get_module_async(callback, context)**
  Gets the `YModule` object for the device on which the function is located (asynchronous version).

- **wakeupmonitor→get_nextWakeUp()**
### 3. Reference

- `get_powerDuration()`
  Returns the next scheduled wake up date/time (UNIX format).

- `get_sleepCountdown()`
  Returns the maximal wake up time (seconds) before automatically going to sleep.

- `get_userData()`
  Returns the delay before the next sleep period.

- `get_wakeUpReason()`
  Returns the value of the userData attribute, as previously stored using method `set_userData`.

- `get_wakeUpState()`
  Returns the latest wake up reason.

- `isOnline()`
  Returns the current state of the monitor.

- `isOnline_async(callback, context)`
  Checks if the monitor is currently reachable, without raising any error (asynchronous version).

- `load(msValidity)`
  Preloads the monitor cache with a specified validity duration.

- `load_async(msValidity, callback, context)`
  Preloads the monitor cache with a specified validity duration (asynchronous version).

- `nextWakeUpMonitor()`
  Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

- `registerValueCallback(callback)`
  Registers the callback function that is invoked on every change of advertised value.

- `resetSleepCountDown()`
  Resets the sleep countdown.

- `set_logicalName(newval)`
  Changes the logical name of the monitor.

- `set_nextWakeUp(newval)`
  Changes the days of the week when a wake up must take place.

- `set_powerDuration(newval)`
  Changes the maximal wake up time (seconds) before automatically going to sleep.

- `set_sleepCountdown(newval)`
  Changes the delay before the next sleep period.

- `set_userData(data)`
  Stores a user context provided as argument in the userData attribute of the function.

- `sleep(secBeforeSleep)`
  Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

- `sleepFor(secUntilWakeUp, secBeforeSleep)`
  Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

- `sleepUntil(wakeUpTime, secBeforeSleep)`
  Goes 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.

- `wait_async(callback, context)`
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wakeupmonitor</code></td>
<td><code>wakeUp()</code></td>
</tr>
<tr>
<td></td>
<td>Forces a wake up.</td>
</tr>
</tbody>
</table>
YWakeUpMonitor.FindWakeUpMonitor()

YWakeUpMonitor.FindWakeUpMonitor()

Retrieves a monitor for a given identifier.

YWakeUpMonitor FindWakeUpMonitor( String func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method YWakeUpMonitor.isOnline() to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters:
- func a string that uniquely characterizes the monitor

Returns:
a YWakeUpMonitor object allowing you to drive the monitor.
YWakeUpMonitor.FirstWakeUpMonitor()

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.
3. Reference

**YWakeUpMonitor**

**wakeupmonitor describe()**

Returns a short text that describes unambiguously the instance of the monitor in the form 
**TYPE**(NAME) = SERIAL.FUNCTIONID.

**String describe()**

More precisely, **TYPE** is the type of the function, **NAME** is the name used for the first access to the function, **SERIAL** is the serial number of the module if the module is connected or "unresolved", and **FUNCTIONID** is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1) = RELAYLO1-123456.relay1 if the module is already connected or Relay(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

wakeupmonitor→get_advertisedValue()
wakeupmonitor→advertisedValue()
wakeupmonitor.get_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

String get_advertisedValue() 

Returns:

- a string corresponding to the current value of the monitor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
YWakeUpMonitor

wakeupmonitor → get_errorMessage()
wakeupmonitor → errorMessage()
wakeupmonitor.get_errorMessage()

Returns the error message of the latest error with the monitor.

String get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the monitor object
**wakeuupmonitor** → **get_errorType()**

**wakeupmonitor** → **errorType()**

**wakeupmonitor.get_errorType()**

Returns the numerical error code of the latest error with the monitor.

```c
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**

a number corresponding to the code of the latest error that occurred while using the monitor object
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.
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```java
String get_functionDescriptor() { }
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**

- an identifier of type `YFUN_DESCR`.

> If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`. 
Returns the hardware identifier of the monitor, without reference to the module.

String `get_functionId()`

For example `relay1`

**Returns:**
- a string that identifies the monitor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
wakeupmonitor→get_hardwareId()  
wakeupmonitor→hardwareId()  
wakeupmonitor.get_hardwareId()  

YWakeUpMonitor

Returns the unique hardware identifier of the monitor in the form SERIAL.FUNCTIONID.

String get_hardwareId()  

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the monitor (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the monitor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
3. Reference

```java
YWakeUpMonitor

wakeupmonitor→get_logicalName()
wakeupmonitor→logicalName()
wakeupmonitor.get_logicalName()
```

Returns the logical name of the monitor.

```java
String get_logicalName()
```

**Returns:**
- a string corresponding to the logical name of the monitor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
get_module() Gets the YModule object for the device on which the function is located.

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>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wakeupmonitor→get_nextWakeUp()</td>
<td>Returns the next scheduled wake up date/time (UNIX format)</td>
</tr>
<tr>
<td>wakeupmonitor→nextWakeUp()</td>
<td></td>
</tr>
<tr>
<td>wakeupmonitor.get_nextWakeUp()</td>
<td></td>
</tr>
</tbody>
</table>

Returns:
- an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns Y_NEXTWAKEUP_INVALID.
wakeupmonitor→get_powerDuration() \hspace{1cm} \text{YWakeUpMonitor}
wakeupmonitor→powerDuration() \hspace{1cm} \text{wakeupmonitor.get_powerDuration()}

Returns the maximal wake up time (in seconds) before automatically going to sleep.

```cpp
int get_powerDuration()
```

Returns:

- an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns \text{Y\_POWERDURATION\_INVALID}. 
wakeupmonitor\rightarrow get\_sleepCountdown() 
YWakeUpMonitor

wakeupmonitor\rightarrow sleepCountdown() 

wakeupmonitor.get\_sleepCountdown()

Returns the delay before the next sleep period.

\begin{verbatim}
int get_sleepCountdown()
\end{verbatim}

\textbf{Returns} :

- an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns Y\_SLEEPCOUNTDOWN\_INVALID.
wakeupmonitor→get_userData()
wakeupmonitor→userData()
wakeupmonitor.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.
Returns the latest wake up reason.

```c
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.
}
```
### YWakeUpMonitor

- `wakeupmonitor.get_wakeUpState()`: Returns the current state of the monitor.

**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`. 

<table>
<thead>
<tr>
<th>YWakeUpMonitor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wakeupmonitor.get_wakeUpState()</code></td>
</tr>
<tr>
<td><code>wakeupmonitor.wakeUpState()</code></td>
</tr>
<tr>
<td><code>int get_wakeUpState()</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wakeupmonitor.get_wakeUpState()</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wakeupmonitor.get_wakeUpState()</code></td>
</tr>
<tr>
<td><code>Returns the current state of the monitor</code></td>
</tr>
<tr>
<td><code>int get_wakeUpState()</code></td>
</tr>
<tr>
<td><code>Returns :</code></td>
</tr>
<tr>
<td><code>either _Y_WAKEUPSTATE_SLEEPING or _Y_WAKEUPSTATE_AWAKE, according to the current state of the monitor</code></td>
</tr>
<tr>
<td><code>On failure, throws an exception or returns _Y_WAKEUPSTATE_INVALID.</code></td>
</tr>
</tbody>
</table>
wakeupmonitor→isOnline()

wakeupmonitor.isOnline()

Checks if the monitor is currently reachable, without raising any error.

boolean isOnline( )

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

Returns:

true if the monitor can be reached, and false otherwise
Preloads the monitor cache with a specified validity duration.

```cpp
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters:**
- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns:**
- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

Returns:
- A pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a null pointer if there are no more monitors to enumerate.
wakeupmonitor.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters**:
- **callback**: the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
wakeupmonitor→resetSleepCountDown() \hspace{1cm} \text{YWakeUpMonitor}
wakeupmonitor.resetSleepCountDown()  

Resets the sleep countdown.

\begin{tabular}{|l|}
\hline
\textbf{int resetSleepCountDown( )} \\
\hline
\end{tabular}

\textbf{Returns:}
\begin{itemize}
\item \textbf{YAPI\_SUCCESS} if the call succeeds. On failure, throws an exception or returns a negative error code.
\end{itemize}
Changes the logical name of the monitor.

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.
YWakeUpMonitor

wakeupmonitor->set_nextWakeUp()
wakeupmonitor->setNextWakeUp()
wakeupmonitor.set_nextWakeUp()

Changes the days of the week when a wake up must take place.

```c
int set_nextWakeUp(long newval)
```

**Parameters:**
- `newval` an integer corresponding to the days of the week when a wake up must take place

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.
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.
YWakeUpMonitor

wakeupmonitor\rightarrow \text{set\_sleepCountdown()}
wakeupmonitor\rightarrow \text{setSleepCountdown()}
wakeupmonitor\. \text{set\_sleepCountdown()}

Changes the delay before the next sleep period.

\begin{Verbatim}
\text{int} \text{set\_sleepCountdown( int newval)}
\end{Verbatim}

**Parameters:**
- \textit{newval} an integer corresponding to the delay before the next sleep period

**Returns:**
- \text{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
wakeupmonitor→sleep()  \[ \text{wakeupmonitor.sleep()} \]  \[ \text{YWakeUpMonitor} \]

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.
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.
3. Reference

**wakeuperonitor→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.

<table>
<thead>
<tr>
<th>Parameters :</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wakeUpTime</code></td>
</tr>
<tr>
<td>wake-up datetime (UNIX format)</td>
</tr>
<tr>
<td><code>secBeforeSleep</code></td>
</tr>
<tr>
<td>number of seconds before going into sleep mode</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.
wakeupmonitor→wakeUp()
wakeupmonitor.wakeUp()

Forces a wake up.

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;
```
```
php
require_once('yocto_wakeupschedule.php');
```
```
cpp
#include "yocto_wakeupschedule.h"
```
```
msg
#import "yocto_wakeupschedule.h"
```
```
pas
uses yocto_wakeupschedule;
```
```
vb
yocto_wakeupschedule.vb
```
```
 cs
yocto_wakeupschedule.cs
```
```
 java
 import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
```
```
 py
from yocto_wakeupschedule import *
```

### Global functions

**yFindWakeUpSchedule(func)**

Retrieves a wake up schedule for a given identifier.

**yFirstWakeUpSchedule()**

Starts the enumeration of wake up schedules currently accessible.

### YWakeUpSchedule methods

**wakeupschedule→describe()**

Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**wakeupschedule→getadvertisedValue()**

Returns the current value of the wake up schedule (no more than 6 characters).

**wakeupschedule→getErrorMessage()**

Returns the error message of the latest error with the wake up schedule.

**wakeupschedule→getErrorType()**

Returns the numerical error code of the latest error with the wake up schedule.

**wakeupschedule→getFriendlyName()**

Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

**wakeupschedule→getFunctionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**wakeupschedule→getFunctionId()**

Returns the hardware identifier of the wake up schedule, without reference to the module.

**wakeupschedule→getHardwareId()**

Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.

**wakeupschedule→getHours()**

Returns the hours scheduled for wake up.

**wakeupschedule→getLogicalName()**

Returns the logical name of the wake up schedule.

**wakeupschedule→getMinutes()**

Returns all the minutes of each hour that are scheduled for wake up.

**wakeupschedule→getMinutesA()**
Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

### wakeupschedule→get_minutesB()
Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

### wakeupschedule→get_module()
Gets the YModule object for the device on which the function is located.

### wakeupschedule→get_module_async(callback, context)
Gets the YModule object for the device on which the function is located (asynchronous version).

### wakeupschedule→get_monthDays()
Returns the days of the month scheduled for wake up.

### wakeupschedule→get_months()
Returns the months scheduled for wake up.

### wakeupschedule→get_nextOccurrence()
Returns the date/time (seconds) of the next wake up occurrence

### wakeupschedule→get_userData()
Returns the value of the userData attribute, as previously stored using method set_userData.

### wakeupschedule→get_weekDays()
Returns the days of the week scheduled for wake up.

### wakeupschedule→isOnline()
Checks if the wake up schedule is currently reachable, without raising any error.

### wakeupschedule→isOnline_async(callback, context)
Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

### wakeupschedule→load(msValidity)
Preloads the wake up schedule cache with a specified validity duration.

### wakeupschedule→load_async(msValidity, callback, context)
Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

### wakeupschedule→nextWakeUpSchedule()
Continues the enumeration of wake up schedules started using yFirstWakeUpSchedule().

### wakeupschedule→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

### wakeupschedule→set_hours(newval)
Changes the hours when a wake up must take place.

### wakeupschedule→set_logicalName(newval)
Changes the logical name of the wake up schedule.

### wakeupschedule→set_minutes(bitmap)
Changes all the minutes where a wake up must take place.

### wakeupschedule→set_minutesA(newval)
Changes the minutes in the 00-29 interval when a wake up must take place.

### wakeupschedule→set_minutesB(newval)
Changes the minutes in the 30-59 interval when a wake up must take place.

### wakeupschedule→set_monthDays(newval)
Changes the days of the month when a wake up must take place.

### wakeupschedule→set_months(newval)
Changes the months when a wake up must take place.

### wakeupschedule→set_userData(data)
Stores a user context provided as argument in the userData attribute of the function.
### wakeupschedule → set_weekDays(newval)
Changes the days of the week when a wake up must take place.

### wakeupschedule → wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.
YWakeUpSchedule.FindWakeUpSchedule()

Retrieves a wake up schedule for a given identifier.

YWakeUpSchedule.FindWakeUpSchedule(String func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method YWakeUpSchedule.isOnline() to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters:**

- **func** a string that uniquely characterizes the wake up schedule

**Returns:**

- a YWakeUpSchedule object allowing you to drive the wake up schedule.
3. Reference

YWakeUpSchedule.FirstWakeUpSchedule()

Starts the enumeration of wake up schedules currently accessible.

Use the method YWakeUpSchedule.nextWakeUpSchedule() to iterate on next wake up schedules.

Returns:
- a pointer to a YWakeUpSchedule object, corresponding to the first wake up schedule currently online,
- or a null pointer if there are none.
YWakeUpSchedule.wakeupschedule.describe()

Returns a short text that describes unambiguously the instance of the wake up schedule in the form TYPE(NAME)=SERIAL.FUNCTIONID.

String describe()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns:

a string that describes the wake up schedule (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)
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.
YWakeUpSchedule

```java
wakeupschedule->get_errorMessage()
```

Returns the error message of the latest error with the wake up schedule.

```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 wake up schedule object
Returns the numerical error code of the latest error with the wake up schedule.

`int get_errorType( )`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
- a number corresponding to the code of the latest error that occurred while using the wake up schedule object
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.
3. Reference

YWakeUpSchedule

wakeupschedule → get_functionDescriptor()

wakeupschedule → functionDescriptor()

wakeupschedule.get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

String `get_functionDescriptor()`

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**
- an identifier of type `YFUN_DESCR`.
- If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`. 
Returns the hardware identifier of the wake up schedule, without reference to the module.

String `get_functionId()`

For example relay1

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a string that identifies the wake up schedule (ex: relay1)</td>
</tr>
</tbody>
</table>

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
YWakeUpSchedule

twakeupSchedule.get_hardwareId()

Returns the unique hardware identifier of the wake up schedule in the form SERIAL.FUNCTIONID.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wake up schedule (for example RELAYLO1-123456.relay1).

Returns:

- a string that uniquely identifies the wake up schedule (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
**wakeupschedule**→**get_hours()**

**wakeupschedule**→**hours()**

**wakeupschedule.get_hours()**

- **Returns the hours scheduled for wake up.**

```c
int get_hours()
```

**Returns:**

- an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns **Y_HOURS_INVALID**.
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`. 
Returns all the minutes of each hour that are scheduled for wake up.

`long get_minutes()`
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.

```c
int get_minutesB()
```

**Returns:**

- an integer corresponding to the minutes in the 30-59 interval of each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESB_INVALID`. 
wakeupschedule->get_module()

YWakeUpSchedule

wakeupschedule->module()

wakeupschedule->get_module()

Gets the YModule object for the device on which the function is located.

YModule get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns:

an instance of YModule
Returns the days of the month scheduled for wake up.

### int get_monthDays()

**Returns:**

- an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns Y_MONTHDAYS_INVALID.
### 3. Reference

**wakeupschedule**→**get_months()**

**wakeupschedule**→**months()**

**wakeupschedule.get_months()**

<table>
<thead>
<tr>
<th>YWakeUpSchedule</th>
</tr>
</thead>
</table>

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`. 
Wakeupschedule → get_nextOccurrence()
Wakeupschedule → nextOccurrence()
Wakeupschedule.get_nextOccurrence()

Returns the date/time (seconds) of the next wake up occurrence

```java
long get_nextOccurrence()
```

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`. 
3. Reference

YWakeUpSchedule

wakeschedule→get_userData()
wakeschedule→userData()
wakeschedule.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.
Returns the days of the week scheduled for wake up.

int get_weekDays() {

Returns:
- an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns Y_WEEKDAYS_INVALID.
YWakeUpSchedule
wakeupschedule.isOnline()  

Checks if the wake up schedule is currently reachable, without raising any error.

boolean isOnline()  

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

Returns:

true if the wake up schedule can be reached, and false otherwise
Preloads the wake up schedule cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
ywakewupschedule\rightarrow\text{nextWakeUpSchedule()}

ywakewupschedule\text{.nextWakeUpSchedule()}

<table>
<thead>
<tr>
<th>YWakeUpSchedule</th>
<th>nextWakeUpSchedule()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Continues the enumeration of wake up schedules started using yFirstWakeUpSchedule().</td>
</tr>
</tbody>
</table>

**Returns:**

- a pointer to a \text{YWakeUpSchedule} object, corresponding to a wake up schedule currently online, or a null pointer if there are no more wake up schedules to enumerate.
### wakeupschedule.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
### 3. Reference

**wakeschedule → set_hours()**

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.
Changes the logical name of the wake up schedule.

### set_logicalName

```java
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.
3. Reference

YWakeUpSchedule

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.
3. Reference

YWakeUpSchedule

wakeupschedule.set_minutesA()

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.
3. Reference

**wakeupschedule->set_minutesB()**

Changes the minutes in the 30-59 interval when a wake up must take place.

<table>
<thead>
<tr>
<th>Parameters :</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newval</code> an integer corresponding to the minutes in the 30-59 interval when a wake up must take place</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.
wakeupschedule->set_monthDays()
wakeupschedule->setMonthDays()
wakeupschedule.set_monthDays()

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
Changes the days of the week when a wake up must take place.

```c
int set_weekDays( int newval)
```

**Parameters:**
- `newval` an integer corresponding to the days of the week when a wake up must take place

**Returns:**
- `YAPI_SUCCESS` if the call succeeds.
- On failure, throws an exception or returns a negative error code.
### 3.49. 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:

```javascript
<script type='text/javascript' src='yocto_watchdog.js'></script>
```

```nodejs
var yoctolib = require('yoctolib');
var YWatchdog = yoctolib.YWatchdog;
require_once('yocto_watchdog.php');
```

```cpp
#include "yocto_watchdog.h"
```

```pas
uses yocto_watchdog;
```

```vb
yocto_watchdog.vb
```

```cs
yocto_watchdog.cs
```

```java
import com.yoctopuce.YoctoAPI.YWatchdog;
```

```py
from yocto_watchdog import *
```

---

#### Global functions

- **yFindWatchdog(func)**
  Retrieves a watchdog for a given identifier.

- **yFirstWatchdog()**
  Starts the enumeration of watchdog currently accessible.

#### YWatchdog methods

- **watchdog→delayedPulse(ms_delay, ms_duration)**
  Schedules a pulse.

- **watchdog→describe()**
  Returns a short text that describes unambiguously the instance of the watchdog in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

- **watchdog→get_advertisedValue()**
  Returns the current value of the watchdog (no more than 6 characters).

- **watchdog→get_autoStart()**
  Returns the watchdog running state at module power on.

- **watchdog→get_countdown()**
  Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

- **watchdog→get_errorMessage()**
  Returns the error message of the latest error with the watchdog.

- **watchdog→get_errorType()**
  Returns the numerical error code of the latest error with the watchdog.

- **watchdog→get_friendlyName()**
  Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.

- **watchdog→get_functionDescriptor()**
  Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

- **watchdog→get_functionId()**
  Returns the hardware identifier of the watchdog, without reference to the module.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>watchdog→get_hardwareId()</strong></td>
<td>Returns the unique hardware identifier of the watchdog in the form <code>SERIAL.FUNCTIONID</code>.</td>
</tr>
<tr>
<td><strong>watchdog→get_logicalName()</strong></td>
<td>Returns the logical name of the watchdog.</td>
</tr>
<tr>
<td><strong>watchdog→get_maxTimeOnStateA()</strong></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><strong>watchdog→get_maxTimeOnStateB()</strong></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><strong>watchdog→get_module()</strong></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located.</td>
</tr>
<tr>
<td><strong>watchdog→get_module_async(callback, context)</strong></td>
<td>Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).</td>
</tr>
<tr>
<td><strong>watchdog→get_output()</strong></td>
<td>Returns the output state of the watchdog, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td><strong>watchdog→get_pulseTimer()</strong></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><strong>watchdog→get_running()</strong></td>
<td>Returns the watchdog running state.</td>
</tr>
<tr>
<td><strong>watchdog→get_state()</strong></td>
<td>Returns the state of the watchdog (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td><strong>watchdog→get_stateAtPowerOn()</strong></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><strong>watchdog→get_triggerDelay()</strong></td>
<td>Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td><strong>watchdog→get_triggerDuration()</strong></td>
<td>Returns the duration of resets caused by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td><strong>watchdog→get_userData()</strong></td>
<td>Returns the value of the userData attribute, as previously stored using method <code>set_userData</code>.</td>
</tr>
<tr>
<td><strong>watchdog→isOnline()</strong></td>
<td>Checks if the watchdog is currently reachable, without raising any error.</td>
</tr>
<tr>
<td><strong>watchdog→isOnline_async(callback, context)</strong></td>
<td>Checks if the watchdog is currently reachable, without raising any error (asynchronous version).</td>
</tr>
<tr>
<td><strong>watchdog→load(msValidity)</strong></td>
<td>Preloads the watchdog cache with a specified validity duration.</td>
</tr>
<tr>
<td><strong>watchdog→load_async(msValidity, callback, context)</strong></td>
<td>Preloads the watchdog cache with a specified validity duration (asynchronous version).</td>
</tr>
<tr>
<td><strong>watchdog→nextWatchdog()</strong></td>
<td>Continues the enumeration of watchdog started using <code>yFirstWatchdog()</code>.</td>
</tr>
<tr>
<td><strong>watchdog→pulse(ms_duration)</strong></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><strong>watchdog→registerValueCallback(callback)</strong></td>
<td>Registers the callback function that is invoked on every change of advertised value.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>watchdog.resetWatchdog()</code></td>
<td>Resets the watchdog.</td>
</tr>
<tr>
<td><code>watchdog.set_autoStart(newval)</code></td>
<td>Changes the watchdog running state at module power on.</td>
</tr>
<tr>
<td><code>watchdog.set_logicalName(newval)</code></td>
<td>Changes the logical name of the watchdog.</td>
</tr>
<tr>
<td><code>watchdog.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>watchdog.set_maxTimeOnStateB(newval)</code></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><code>watchdog.set_output(newval)</code></td>
<td>Changes the output state of the watchdog, when used as a simple switch (single throw).</td>
</tr>
<tr>
<td><code>watchdog.set_running(newval)</code></td>
<td>Changes the running state of the watchdog.</td>
</tr>
<tr>
<td><code>watchdog.set_state(newval)</code></td>
<td>Changes the state of the watchdog (A for the idle position, B for the active position).</td>
</tr>
<tr>
<td><code>watchdog.set_stateAtPowerOn(newval)</code></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><code>watchdog.set_triggerDelay(newval)</code></td>
<td>Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td><code>watchdog.set_triggerDuration(newval)</code></td>
<td>Changes the duration of resets caused by the watchdog, in milliseconds.</td>
</tr>
<tr>
<td><code>watchdog.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>watchdog.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>
YWatchdog.FindWatchdog() YWatchdog

Retrieves a watchdog for a given identifier.

YWatchdog FindWatchdog(String func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method YWatchdog.isOnline() to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters:**
- **func** a string that uniquely characterizes the watchdog

**Returns:**
- a YWatchdog object allowing you to drive the watchdog.
**YWatchdog.FirstWatchdog()**

**yFirstWatchdog()**

YWatchdog.FirstWatchdog()

```c

Returns:

A pointer to a YWatchdog object, corresponding to the first watchdog currently online, or a null pointer if there are none.
```
## Reference

**YWatchdog**

**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.
**Watchdog.describe()**

Returns a short text that describes unambiguously the instance of the watchdog 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 watchdog (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)
Returns the current value of the watchdog (no more than 6 characters).

String `get_advertisedValue()`

**Returns:**
- a string corresponding to the current value of the watchdog (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`. 
YWatchdog

YWatchdog

Returns the watchdog running state at module power on.

int get_autoStart()

Returns:

- either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog running state at module power on

On failure, throws an exception or returns Y_AUTOSTART_INVALID.
YWatchdog

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>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>countdown()</code></td>
<td></td>
</tr>
</tbody>
</table>
Returns the error message of the latest error with the watchdog.

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
YWatchdog

```plaintext
watchdog.get_errorType()
watchdog.errorType()
watchdog.get_errorType()
```

Returns the numerical error code of the latest error with the watchdog.

```plaintext
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns:
- a number corresponding to the code of the latest error that occurred while using the watchdog object
**watchdog.get_friendlyName()**

Returns a global identifier of the watchdog in the format **MODULE_NAME.FUNCTION_NAME**.

**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.

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`.
YWatchdog

3. Reference

```java
YWatchdog
watchdog.get_functionId()
```

Returns the hardware identifier of the watchdog, without reference to the module.

```java
String get_functionId()
```

For example `relay1`

**Returns:**

- a string that identifies the watchdog (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`. 
YWatchdog

watchdog→get_hardwareId()
watchdog→hardwareId()
watchdog.get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog (for example RELAYLO1-123456.relay1).

Returns :

- a string that uniquely identifies the watchdog (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
Returns the logical name of the watchdog.

String `get_logicalName()`

**Returns:**

- a string corresponding to the logical name of the watchdog.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`. 
Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state $A$ before automatically switching back to state $B$.

**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.

```c
long get_maxTimeOnStateB( )
```

Zero means no maximum time.

**Returns:**

- an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.
YWatchdog

**watchdog.get_module()**

```
YModule get_module()
```

**YModule module()**

```
module()
```

```ywatchdog
watchdog.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
Returns the output state of the watchdog, when used as a simple switch (single throw).

int get_output()

Returns:
- either \texttt{Y\_OUTPUT\_OFF} or \texttt{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 \texttt{Y\_OUTPUT\_INVALID}. 
3. Reference

```plaintext
YWatchdog
watchdog→get_pulseTimer()
watchdog→pulseTimer()
watchdog.get_pulseTimer()
```

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

```plaintext
long get_pulseTimer()
```

When there is no ongoing pulse, returns zero.

**Returns:**

- an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`. 
### YWatchdog

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_running()</code></td>
<td>Returns the watchdog running state.</td>
</tr>
<tr>
<td><code>running()</code></td>
<td>Returns: either Y_RUNNING_OFF or Y_RUNNING_ON, according to the watchdog running state.</td>
</tr>
<tr>
<td></td>
<td>On failure, throws an exception or returns Y_RUNNING_INVALID.</td>
</tr>
</tbody>
</table>

```c
int get_running()
```

```c
YWatchdog
```
```c
watchdog->get_running()
```
```c
watchdog->running()
```
```c
watchdog.get_running()
```
### 3. Reference

<table>
<thead>
<tr>
<th>YWatchdog</th>
<th>watchdog.get_state()</th>
<th>watchdog.state()</th>
<th>watchdog.get_state()</th>
</tr>
</thead>
</table>

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`. 
**YWatchdog**

**watchdog.get_stateAtPowerOn()**

**YWatchdog**

**watchdog.stateAtPowerOn()**

**watchdog.get_stateAtPowerOn()**

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```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.

**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

`YWatchdog.get_triggerDuration()`

Returns the duration of resets caused by the watchdog, in milliseconds.

```
long get_triggerDuration( )
```

**Returns:**

- an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDURATION_INVALID`. 
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.
YWatchdog

\texttt{watchdog\→isOnline()} \texttt{watchdog\ .isOnline()} \texttt{YWatchdog}

Checks if the watchdog is currently reachable, without raising any error.

\texttt{boolean isOnline()}

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

\textbf{Returns}:
\begin{itemize}
  \item \texttt{true} if the watchdog can be reached, and \texttt{false} otherwise
\end{itemize}
Preloads the watchdog 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.
YWatchdog
nextWatchdog()  
Continues the enumeration of watchdog started using yFirstWatchdog().

YWatchdog nextWatchdog()  

Returns:
- a pointer to a YWatchdog object, corresponding to a watchdog currently online, or a null pointer if there are no more watchdog to enumerate.
### YWatchdog

**watchdog.pulse()**

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.
3. Reference

**YWatchdog**

`watchdog.registerValueCallback()`

Registers the callback function that is invoked on every change of advertised value.

```c
int registerValueCallback( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**

- `callback` the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
watchdog\rightarrow \text{resetWatchdog()}

\text{resetWatchdog()}$

Resets the watchdog.

\begin{Verbatim}
\textbf{int resetWatchdog( )}
\end{Verbatim}

When the watchdog is running, this function must be called on a regular basis to prevent the watchdog to trigger

\begin{Verbatim}
\textbf{Returns :}
\end{Verbatim}

\begin{Verbatim}
\textbf{YAPI\_SUCCESS} if the call succeeds.
\end{Verbatim}

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.
3. Reference

**YWatchdog**

```java
watchdog.set_logicalName()
```

Changes the logical name of the watchdog.

### 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.

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.
YWatchdog

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.
3. Reference

YWatchdog

watchdog→set_maxTimeOnStateB()
watchdog→setMaxTimeOnStateB()
watchdog.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```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).

**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.
### YWatchdog

**watchdog→set_state()**

Changes the state of the watchdog (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 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.
3. Reference

```c
int set_stateAtPowerOn( int newval)
```

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

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.
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.
**watchdog.set_userData()**

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
3.50. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

```javascript
<script type='text/javascript' src='yocto_wireless.js'></script>
```
```nodejs
var yoctolib = require('yoctolib');
var YWireless = yoctolib.YWireless;
```
```php
require_once('yocto_wireless.php');
```
```cpp
#include "yocto_wireless.h"
```
```m
#include "yocto_wireless.h"
```
```pas
uses yocto_wireless;
```
```vb
yocto_wireless.vb
```
```cs
yocto_wireless.cs
```
```java
import com.yoctopuce.YoctoAPI.YWireless;
```
```py
from yocto_wireless import *
```

## Global functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yFindWireless(func)</code></td>
<td>Retrieves a wireless lan interface for a given identifier.</td>
</tr>
<tr>
<td><code>yFirstWireless()</code></td>
<td>Starts the enumeration of wireless lan interfaces currently accessible.</td>
</tr>
</tbody>
</table>

## YWireless methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wireless→adhocNetwork(ssid, securityKey)</code></td>
<td>Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.</td>
</tr>
<tr>
<td><code>wireless→describe()</code></td>
<td>Returns a short text that describes unambiguously the instance of the wireless lan interface in the form TYPE(NAME)=SERIAL.FUNCTIONID.</td>
</tr>
<tr>
<td><code>wireless→get_advertisedValue()</code></td>
<td>Returns the current value of the wireless lan interface (no more than 6 characters).</td>
</tr>
<tr>
<td><code>wireless→get_channel()</code></td>
<td>Returns the 802.11 channel currently used, or 0 when the selected network has not been found.</td>
</tr>
<tr>
<td><code>wireless→get_detectedWlans()</code></td>
<td>Returns a list of YWlanRecord objects that describe detected Wireless networks.</td>
</tr>
<tr>
<td><code>wireless→get_errorMessage()</code></td>
<td>Returns the error message of the latest error with the wireless lan interface.</td>
</tr>
<tr>
<td><code>wireless→get_errorType()</code></td>
<td>Returns the numerical error code of the latest error with the wireless lan interface.</td>
</tr>
<tr>
<td><code>wireless→get_friendlyName()</code></td>
<td>Returns a global identifier of the wireless lan interface in the format MODULE_NAME.FUNCTION_NAME.</td>
</tr>
<tr>
<td><code>wireless→get_functionDescriptor()</code></td>
<td>Returns a unique identifier of type YFUN_DESCR corresponding to the function.</td>
</tr>
<tr>
<td><code>wireless→get_functionId()</code></td>
<td>Returns the hardware identifier of the wireless lan interface, without reference to the module.</td>
</tr>
<tr>
<td><code>wireless→get_hardwareId()</code></td>
<td>Returns the unique hardware identifier of the wireless lan interface in the form SERIAL.FUNCTIONID.</td>
</tr>
</tbody>
</table>
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.
YWireless.FindWireless()

YWireless.FindWireless() retrieves a wireless lan interface for a given identifier.

YWireless.FindWireless(String func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wireless lan interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method YWireless.isOnline() to test if the wireless lan interface is indeed online at a given time. In case of ambiguity when looking for a wireless lan interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters:

- **func**: a string that uniquely characterizes the wireless lan interface

Returns:

- a YWireless object allowing you to drive the wireless lan interface.
YWireless.FirstWireless()

YWireless.FirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

Use the method YWireless.nextWireless() to iterate on next wireless lan interfaces.

Returns:
- a pointer to a YWireless object, corresponding to the first wireless lan interface currently online, or a null pointer if there are none.
wireless→adhocNetwork()
wireless.adhocNetwork()

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

```java
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()\n
Returns a short text that describes unambiguously the instance of the wireless lan interface in the form \texttt{TYPE (NAME) = SERIAL.FUNCTIONID}.

\texttt{String describe( )}

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.

\textbf{Returns :}

\texttt{a string that describes the wireless lan interface (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)}
### 3. Reference

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wireless.get_advertisedValue()</code></td>
<td>Returns the current value of the wireless lan interface (no more than 6 characters).</td>
</tr>
<tr>
<td><code>wireless.advertisedValue()</code></td>
<td>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 <code>Y_ADVERTISEDVALUE_INVALID</code>.</td>
</tr>
</tbody>
</table>

**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`. 
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wireless.get_channel()</code></td>
<td>Returns the 802.11 channel currently used, or 0 when the selected network has not been found.</td>
</tr>
</tbody>
</table>

**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`.  

YWireless
3. Reference

wireless→get_detectedWlans() YWireless
wireless→detectedWlans()
wireless.get_detectedWlans()  

Returns a list of YWlanRecord objects that describe detected Wireless networks.

ArrayList<YWlanRecord> get_detectedWlans()

This list is not updated when the module is already connected to an access point (infrastructure mode). To force an update of this list, adhocNetwork() must be called to disconnect the module from the current network. The returned list must be unallocated by the caller.

Returns:
- a list of YWlanRecord objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.
wireless.get_errorMessage()  
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
YWireless
wireless.get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

```c
int get_errorType()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns:**
- a number corresponding to the code of the latest error that occurred while using the wireless lan interface object
Returns a global identifier of the wireless lan interface in the format
MODULE_NAME.FUNCTION_NAME.

String get_friendlyName()

The returned string uses the logical names of the module and of the wireless lan interface if they are
defined, otherwise the serial number of the module and the hardware identifier of the wireless lan
interface (for example: MyCustomName.relay1)

Returns:
- a string that uniquely identifies the wireless lan interface using logical names (ex: MyCustomName.relay1)
- On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.
wireless → `get_functionDescriptor()`
wireless → `functionDescriptor()`
wireless.get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**String get_functionDescriptor()**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns:**

- an identifier of type `YFUN_DESCR`.

If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTION_INVALID`.
wireless\rightarrow get\_functionId()
wireless\rightarrow functionId()
wireless.get\_functionId()

YWireless

Returns the hardware identifier of the wireless lan interface, without reference to the module.

String get\_functionId()

For example relay1

**Returns** :

- a string that identifies the wireless lan interface (ex: relay1)

On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.
wireless→get_hardwareId()
wireless→hardwareId()
wireless.get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form SERIAL.FUNCTIONID.

String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wireless lan interface (for example RELAYLO1-123456.relay1).

Returns:

- a string that uniquely identifies the wireless lan interface (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.
wireless->get_linkQuality()
wireless->linkQuality()
wireless.get_linkQuality()

Returns the link quality, expressed in percent.

int get_linkQuality()

Returns:
- an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns Y_LINKQUALITY_INVALID.
Returns the logical name of the wireless lan interface.

String \texttt{get\_logicalName()} \(\rightarrow\) returns a string corresponding to the logical name of the wireless lan interface.

On failure, throws an exception or returns \texttt{Y\_LOGICALNAME\_INVALID}. 
wireless → get_message()
wireless → message() wireless.get_message()

Returns the latest status message from the wireless interface.

String get_message()

Returns:

- a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns Y_MESSAGE_INVALID.
3. Reference

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()
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 \text{get\_ssid()}
wireless\rightarrow \text{ssid()}
wireless.get\_ ssid()

Returns the wireless network name (SSID).

\textbf{String} \textit{get\_ssid()} \\

\textbf{Returns}:

- a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns \texttt{Y\_SSID\_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.

```
Object get_userData()
```
### Reference

**wireless→isOnline() wireless.isOnline()**

Checks if the wireless lan interface is currently reachable, without raising any error.

```java
boolean isOnline()
```

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

**Returns:**

- `true` if the wireless lan interface can be reached, and `false` otherwise
wireless→joinNetwork()wireless.joinNetwork()  

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

```java
int joinNetwork(String ssid, String securityKey)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssid</td>
<td>the name of the network to connect to</td>
</tr>
<tr>
<td>securityKey</td>
<td>the network key, as a character string</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.
Preloads the wireless lan interface cache with a specified validity duration.

```c
int load( long msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

- `msValidity` an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

- `YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.
wireless\rightarrow \text{nextWireless()}
wireless.nextWireless()

Continues the enumeration of wireless lan interfaces started using \text{yFirstWireless()}.

\text{YWireless nextWireless( )}

\textbf{Returns}:

a pointer to a \text{YWireless} object, corresponding to a wireless lan interface currently online, or a \text{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( UpdateCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters:**
- **callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.
Changes the logical name of the wireless lan interface.

```java
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.
3. Reference

wireless→set_userData()
wireless→setUserData()wireless.set_userData()

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
wireless softenedAPNetwork()

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

```java
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 36
Access 3
Activating 4
adhocNetwork, YWireless 1763
Altitude 78
AnButton 120
Android 3, 4

B
Blueprint 16
brakingForceMove, YMotor 876

C
calibrate, YLightSensor 746
calibrateFromPoints, YAccelerometer 40
calibrateFromPoints, YAltitude 82
calibrateFromPoints, YCarbonDioxide 162
calibrateFromPoints, YCompass 230
calibrateFromPoints, YCurrent 270
calibrateFromPoints, YGenericSensor 555
calibrateFromPoints, YGyro 604
calibrateFromPoints, YHumidity 680
calibrateFromPoints, YLightSensor 747
calibrateFromPoints, YMagnetometer 788
calibrateFromPoints, YPower 997
calibrateFromPoints, YPressure 1040
calibrateFromPoints, YPwmInput 1079
calibrateFromPoints, YQt 1188
calibrateFromPoints, YSensor 1326
calibrateFromPoints, YTemperature 1457
calibrateFromPoints, YTilt 1498
calibrateFromPoints, YVoc 1537
calibrateFromPoints, YVoltage 1576
callbackLogin, YNetwork 918
cancel3DCalibration, YRefFrame 1254
CarbonDioxide 158
checkFirmware, YModule 836
CheckLogicalName, YAPI 18
clear, YDisplayLayer 468
clearConsole, YDisplayLayer 469
Clock 1223
ColorLed 197
Compass 226
Compatibility 3
Configuration 1250
consoleOut, YDisplayLayer 470
copyLayerContent, YDisplay 424
Current 266

D
Data 339, 349, 361

DataLogger 305
delayedPulse, YDigitalIO 380
delayedPulse, YRelay 1290
delayedPulse, YWatchdog 1719
describe, YAccelerometer 41
describe, YAltitude 83
describe, YAnButton 124
describe, YCarbonDioxide 163
describe, YColorLed 200
describe, YCompass 231
describe, YCurrent 271
describe, YDataLogger 309
describe, YDigitalIO 381
describe, YDisplay 425
describe, YDualPower 502
describe, YFiles 527
describe, YGenericSensor 556
describe, YGyro 605
describe, YHubPort 654
describe, YHumidity 681
describe, YLed 718
describe, YLightSensor 748
describe, YMagnetometer 789
describe, YModule 837
describe, YMotor 877
describe, YNetwork 919
describe, YOsControl 973
describe, YPower 998
describe, YPressure 1041
describe, YPwmInput 1080
describe, YPwmOutput 1127
describe, YPwmPowerSource 1164
describe, YQt 1189
describe, YRealTimeClock 1226
describe, YRefFrame 1255
describe, YRelay 1291
describe, YSensor 1327
describe, YSerialPort 1366
describe, YServo 1422
describe, YTemperature 1458
describe, YTilt 1499
describe, YVoc 1538
describe, YVoltage 1577
describe, YVSource 1614
describe, YWakeUpMonitor 1647
describe, YWakeUpSchedule 1682
describe, YWatchdog 1720
describe, YWireless 1764

Digital 376
Display 420
DisplayLayer 467
drawBar, YDisplayLayer 471
drawBitmap, YDisplayLayer 472
drawCircle, YDisplayLayer 473
drawDisc, YDisplayLayer 474
drawImage, YDisplayLayer 475
drawPixel, YDisplayLayer 476
drawRect, YDisplayLayer 477
drawText, YDisplayLayer 478
drivingForceMove, YMotor 878
dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

drawImage, YDisplayLayer 475

drawPixel, YDisplayLayer 476

drawRect, YDisplayLayer 477

drawText, YDisplayLayer 478

drivingForceMove, YMotor 878

dutyCycleMove, YPwmOutput 1128

E

EnableUSBHost, YAPI 19
Error 13
External 499

F

fade, YDisplay 426
Files 524
FindAccelerometer, YAccelerometer 38
FindAltitude, YAltitude 80
FindAnButton, YAnButton 122
FindCarbonDioxide, YCarbonDioxide 160
FindColorLed, YColorLed 198
FindCompass, YCompass 228
FindCurrent, YCurrent 268
FindDataLogger, YDataLogger 307
FindDisplay, YDisplay 422
FindDualPower, YDualPower 500
FindFiles, YFiles 525
FindGenericSensor, YGenericSensor 553
FindGyro, YGyro 602
FindHubPort, YHubPort 652
FindHumidity, YHumidity 678
FindLed, YLed 716
FindLightSensor, YLightSensor 744
FindMagnetometer, YMagnetometer 786
FindModule, YModule 834
FindMotor, YMotor 874
FindNetwork, YNetwork 916
FindOsControl, YOsControl 971
FindPower, YPower 995
FindPressure, YPressure 1038
FindPwmInput, YPwmInput 1077
FindPwmOutput, YPwmOutput 1125
FindPwmPowerSource, YPwmPowerSource 1162
FindQt, YQt 1186
FindRealTimeClock, YRealTimeClock 1225
FindRefFrame, YRefFrame 1253
FindRelay, YRelay 1289
FindSensor, YSensor 1325
FindSerialPort, YSerialPort 1365
FindServo, YServo 1421
FirstTemp, YTilt 1497
FirstVoc, YVoc 1536
FirstVoltage, YVoltage 1575
FirstVSource, YVSource 1613
FirstWakeUpMonitor, YWakeUpMonitor 1646
FirstWakeUpSchedule, YWakeUpSchedule 1681
FirstWatchdog, YWatchdog 1718
FirstWireless, YWireless 1762
forgetAllDataStreams, YDataLogger 310
format_fs, YFiles 528
Formatted 339
Frame 1250
FreeAPI, YAPI 20
Functions 17

G

General 17
GenericSensor 551
get_3DCalibrationHint, YRefFrame 1256
get_3DCalibrationLogMsg, YRefFrame 1257
get_3DCalibrationProgress, YRefFrame 1258
get_3DCalibrationStage, YRefFrame 1259
get_3DCalibrationStageProgress, YRefFrame 1260
get_adminPassword, YNetwork 920
get_advertisedValue, YAccelerometer 42
get_advertisedValue, YAltitude 84
get_advertisedValue, YAnButton 125
get_advertisedValue, YCarbonDioxide 164
get_advertisedValue, YColorLed 201
get_advertisedValue, YCompass 232
get_advertisedValue, YCurrent 272
get_advertisedValue, YDataLogger 311
get_advertisedValue, YDigitalIO 382
get_advertisedValue, YDisplay 427
get_advertisedValue, YDualPower 503
get_advertisedValue, YFiles 529
get_advertisedValue, YGenericSensor 557
get_advertisedValue, YGyro 606
get_advertisedValue, YHubPort 655
get_advertisedValue, YHumidity 682
get_advertisedValue, YLed 719
get_advertisedValue, YLightSensor 749
get_advertisedValue, YMagnetometer 790
get_advertisedValue, YMotor 879
get_advertisedValue, YNetwork 921
get_advertisedValue, YOsControl 974
get_advertisedValue, YPower 999
get_advertisedValue, YPressure 1042
get_advertisedValue, YPwmInput 1081
get_advertisedValue, YPwmOutput 1129
get_advertisedValue, YPwmPowerSource 1165
get_advertisedValue, YQt 1190
get_advertisedValue, YRealTimeClock 1227
get_advertisedValue, YRefFrame 1261
get_advertisedValue, YRelay 1292
get_advertisedValue, YSensor 1328
get_advertisedValue, YSerialPort 1368
get_advertisedValue, YServo 1423
get_advertisedValue, YTilt 1459
get_advertisedValue, YTilt 1500
get_advertisedValue, YVoc 1539
get_advertisedValue, YVoltage 1578
get_advertisedValue, YVSource 1615
get_advertisedValue, YWakeUpMonitor 1648
get_advertisedValue, YWakeUpSchedule 1683
get_advertisedValue, YWatchdog 1721
get_advertisedValue, YWireless 1765
get_analogCalibration, YAnButton 126
get_autoStart, YDataLogger 312
get_autoStart, YWatchdog 1722
get_averageValue, YDataRun 339
get_averageValue, YDataStream 362
get_averageValue, YMeasure 826
get_baudRate, YHubPort 656
get_beacon, YModule 838
get_beaconDriven, YDataLogger 313
get_bearing, YRefFrame 1262
get_bitDirection, YDigitalIO 383
get_currentValue, YVoc 1541
get_currentValue, YVoltage 1580
get_cutOffVoltage, YMotor 881
get_data, YDataStream 365
get_dataRows, YDataStream 366
get_dataSamplesIntervalMs, YDataStream 367
get_dataSets, YDataLogger 315
get_dataStreams, YDataLogger 316
get_dateTime, YRealTimeClock 1228
get_detectedWlans, YWireless 1767
get_discoverable, YNetwork 928
get_display, YDisplayLayer 479
get_displayHeight, YDisplayLayer 480
get_displayType, YDisplayLayer 481
get_displayWidth, YDisplayLayer 482
get_drivingForce, YMotor 882
get_duration, YDataRun 340
get_duration, YDataStream 368
get_dutyCycle, YPwmInput 1084
get_dutyCycle, YPwmOutput 1130
get_dutyCycleAtPowerOn, YPwmOutput 1131
get_enabled, YDisplay 433
get_enabled, YHubPort 657
get_enabled, YPwmOutput 1132
get_enabled, YServo 1424
get_enabledAtPowerOn, YPwmOutput 1133
get_enabledAtPowerOn, YServo 1425
get_endTimeUTC, YDataSet 350
get_endTimeUTC, YMeasure 827
get_errCount, YSerialPort 1369
get_errorMessage, YAccelerometer 45
get_errorMessage, YAltitude 87
get_errorMessage, YAnButton 130
get_errorMessage, YCarbonDioxide 167
get_errorMessage, YCompass 235
get_errorMessage, YCurrent 275
get_errorMessage, YDataLogger 317
get_errorMessage, YDisplay 434
get_errorMessage, YDualPower 504
get_errorMessage, YFiles 530
get_errorMessage, YGenericSensor 560
get_errorMessage, YGyro 609
get_errorMessage, YHubPort 658
get_errorMessage, YHumidity 685
get_errorMessage, YLed 721
get_errorMessage, YLightSensor 752
get_errorMessage, YModule 839
get_errorMessage, YMotor 883
get_errorMessage, YNetwork 929
get_errorMessage, YOsControl 975
get_errorMessage, YPower 1003
get_errorMessage, YPressure 1045
get_errorMessage, YPwmInput 1085
get_errorMessage, YPwmOutput 1134
get_errorMessage, YPwmPowerSource 1166
get_errorMessage, YQt 1193
get_errorMessage, YRealTimeClock 1229
get_errorMessage, YRefFrame 1263
get_errorMessage, YRelay 1294
get_errorMessage, YSensor 1331
get_errorMessage, YSerialPort 1370
get_errorMessage, YServo 1426
get_errorMessage, YTemperature 1462
get_errorMessage, YTilt 1503
get_errorMessage, YVs 1542
get_errorMessage, YVs 1581
get_errorMessage, YVs 1616
get_errorMessage, YWakeUpMonitor 1649
get_errorMessage, YWakeUpSchedule 1684
get_errorMessage, YWatchdog 1724
get_errorMessage, YWireless 1768
get_errorType, YAccelerometer 46
get_errorType, YAltitude 88
get_errorType, YAnButton 131
get_errorType, YCarbonDioxide 168
get_errorType, YColorLed 203
get_errorType, YCompass 236
get_errorType, YCurrent 276
get_errorType, YDataLogger 318
get_errorType, YDisplay 435
get_errorType, YDigitalIO 388
get_errorType, YDualPower 505
get_errorType, YFiles 531
get_errorType, YGenericSensor 561
get_errorType, YGyro 610
get_errorType, YHubPort 659
get_errorType, YHumidity 686
get_errorType, YLed 722
get_errorType, YLightSensor 753
get_errorType, YModule 840
get_errorType, YMotor 884
get_errorType, YNetwork 930
get_errorType, YOsControl 976
get_errorType, YPower 1004
get_errorType, YPressure 1046
get_errorType, YPwmIn 1086
get_errorType, YPwmOut 1135
get_errorType, YPwmPowerSource 1167
get_errorType, YQt 1194
get_errorType, YRealTimeClock 1230
get_errorType, YRefFrame 1264
get_errorType, YRelay 1295
get_errorType, YSensor 1332
get_errorType, YSerialPort 1371
get_errorType, YServo 1427
get_errorType, YTemperature 1463
get_errorType, YTilt 1504
get_errorType, YVs 1543
get_errorType, YVs 1582
get_errorType, YVs 1617
get_errorType, YWakeUpMonitor 1650
get_functionId, YMagnetometer 797
get_functionId, YMotor 889
get_functionId, YNetwork 933
get_functionId, YOsControl 979
get_functionId, YPower 1007
get_functionId, YPressure 1049
get_functionId, YPwmInput 1090
get_functionId, YPwmOutput 1139
get_functionId, YPwmPowerSource 1170
get_functionId, YQt 1197
get_functionId, YRealTimeClock 1233
get_functionId, YRefFrame 1267
get_functionId, YRelay 1298
get_functionId, YSensor 1335
get_functionId, YSerialPort 1374
get_functionId, YServo 1430
get_functionId, YTilt 1507
get_functionId, YVoltage 1585
get_functionId, YVSource 1623
get_functionId, YWakeUpMonitor 1654
get_functionId, YWakeUpSchedule 1689
get_functionId, YWatchdog 1729
get_functionId, YWireless 1773
get_hardwareId, YMagnetometer 50
get_hardwareId, YMotor 890
get_hardwareId, YNetwork 934
get_hardwareId, YOsControl 980
get_hardwareId, YPower 1008
get_hardwareId, YPressure 1050
get_hardwareId, YPwmInput 1091
get_hardwareId, YPwmOutput 1140
get_hardwareId, YPwmPowerSource 1171
get_hardwareId, YQt 1198
get_hardwareId, YRealTimeClock 1234
get_hardwareId, YRefFrame 1268
get_hardwareId, YRelay 1299
get_hardwareId, YSensor 1336
get_hardwareId, YSerialPort 1375
get_hardwareId, YServo 1431
get_hardwareId, YTemperature 1467
get_hardwareId, YTilt 1508
get_hardwareId, YVoc 1547
get_hardwareId, YVoltage 1586
get_hardwareId, YVoltageSource 1623
get_hardwareId, YWakeUpMonitor 1654
get_hardwareId, YWakeUpSchedule 1689
get_hardwareId, YWatchdog 1729
get_hardwareId, YWireless 1773
get_highestValue, YAccelerometer 51
get_highestValue, YAltitude 93
get_highestValue, YCarbonDioxide 173
get_highestValue, YCompass 241
get_highestValue, YCurrent 281
get_highestValue, YGenericSensor 566
get_highestValue, YGyro 616
get_highestValue, YHumidity 691
get_highestValue, YLightSensor 758
get_highestValue, YMagnetometer 799
get_highestValue, YPower 1009
get_highestValue, YPressure 1051
get_highestValue, YPwmInput 1092
get_highestValue, YQt 1199
get_highestValue, YSensor 1337
get_highestValue, YTemperature 1468
get_highestValue, YTilt 1509
get_highestValue, YVoltage 1587
get_hours, YWakeUpSchedule 1690
get_hslColor, YColorLed 208
get_ipAddress, YNetwork 935
get_isPressed, YAnButton 136
get_lastLogs, YModule 843
get_lastMsg, YSerialPort 1376
get_lastTimePressed, YAnButton 137
get_lastTimeReleased, YAnButton 138
get_layerCount, YDisplay 440
get_layerHeight, YDisplay 441
get_layerHeight, YDisplayLayer 482
get_layerWidth, YDisplay 442
get_layerWidth, YDisplayLayer 483
get_linkQuality, YWireless 1774
get_list, YFiles 538
get_logFrequency, YAccelerometer 52
get_logFrequency, YAltitude 94
get_logFrequency, YCarbonDioxide 174
get_logFrequency, YCompass 242
get_logFrequency, YCurrent 282
get_logFrequency, YGenericSensor 567
get_logFrequency, YGyro 617
get_logFrequency, YHumidity 692
get_logFrequency, YLightSensor 759
get_logFrequency, YMagnetometer 800
get_logFrequency, YPressure 1052
get_logFrequency, YPower 1010
get_resolution, YAltitude 101
get_resolution, YCarbonDioxide 180
get_resolution, YCompass 249
get_resolution, YCurrent 288
get_resolution, YGenericSensor 573
get_resolution, YGyro 628
get_resolution, YHumidity 698
get_resolution, YLightSensor 766
get_resolution, YMagnetometer 806
get_resolution, YPower 1018
get_resolution, YPressure 1058
get_resolution, YPwmInput 1104
get_resolution, YQt 1206
get_resolution, YSensor 1344
get_resolution, YTemperature 1475
get_resolution, YTilt 1516
get_resolution, YVoc 1555
get_resolution, YVoltage 1594
get_rgbColor, YColorLed 211
get_rgbColorAtPowerOn, YColorLed 212
get_roll, YGyro 629
get_router, YNetwork 942
get_rowCount, YDataStream 371
get_runIndex, YDataStream 372
get_running, YWatchdog 1736
get_secondaryDNS, YNetwork 943
get_security, YWireless 1778
get_sensitivity, YAnButton 144
get_sensorType, YTemperature 1476
get_serialMode, YSerialPort 1382
get_serialNumber, YModule 851
get_shutdownCountdown, YOsControl 983
get_signalBias, YGenericSensor 574
get_signalRange, YGenericSensor 575
get_signalUnit, YGenericSensor 576
get_signalValue, YGenericSensor 577
get_sleepCountdown, YWakeUpMonitor 1659
get_sshd, YWireless 1779
get_state, YRelay 1306
get_state, YWatchdog 1737
get_stateAtPowerOn, YRelay 1307
get_stateAtPowerOn, YWatchdog 1738
get_subnetMask, YNetwork 944
get_summary, YDataRun 357
get_timeSet, YRealTimeClock 1237
get_timeUTC, YDataLogger 326
get_triggerDelay, YWatchdog 1739
get_triggerDuration, YWatchdog 1740
get_txCOUNT, YSerialPort 1383
get_unit, YAccelerometer 59
get_unit, YAltitude 102
get_unit, YCarbonDioxide 181
get_unit, YCompass 250
get_unit, YCurrent 289
get_unit, YDataSet 358
get_unit, YGenericSensor 578
get_unit, YGyro 630
get_unit, YHumidity 699
get_unit, YLightSensor 767
get_unit, YMagnetometer 807
get_unit, YPower 1019
get_unit, YPressure 1059
get_unit, YPwmInput 1105
get_unit, YQt 1207
get_unit, YSensor 1345
get_unit, YTemperature 1477
get_unit, YTilt 1517
get_unit, YVoc 1556
get_unit, YVoltage 1595
get_unit, YVSource 1630
get_unixTime, YRealTimeClock 1238
get_upTime, YModule 852
get_usbCurrent, YModule 853
get_userData, YAccelerometer 60
get_userData, YAltitude 103
get_userData, YAnButton 145
get_userData, YCarbonDioxide 182
get_userData, YColorLed 213
get_userData, YCompass 251
get_userData, YCurrent 290
get_userData, YDataLogger 327
get_userData, YDigitalIO 401
get_userData, YDisplay 447
get_userData, YDualPower 515
get_userData, YFiles 541
get_userData, YGenericSensor 579
get_userData, YGyro 631
get_userData, YHubPort 667
get_userData, YHumidity 700
get_userData, YLed 731
get_userData, YLightSensor 768
get_userData, YMagnetometer 808
get_userData, YModule 854
get_userData, YMotor 896
get_userData, YNetwork 945
get_userData, YOsControl 984
get_userData, YPower 1020
get_userData, YPressure 1060
get_userData, YPwmInput 1106
get_userData, YPwmOutput 1145
get_userData, YPwmPowerSource 1175
get_userData, YQt 1208
get_userData, YRealTimeClock 1239
get_userData, YRefFrame 1273
get_userData, YRelay 1308
get_userData, YSensor 1346
get_userData, YSerialPort 1384
get_userData, YServo 1438
get_userData, YTemperature 1478
get_userData, YTilt 1518
get_userData, YVoc 1557
get_userData, YVoltage 1596
get_userData, YVSource 1631
get_userData, YWakeUpMonitor 1660
get_userData, YWakeUpSchedule 1699
get_userData, YWatchdog 1741
get_userData, YWireless 1780
get_userPassword, YNetwork 946
get_userVar, YModule 855
get_utcOffset, YRealTimeClock 1240
get_valueCount, YDataRun 345
get_valueInterval, YDataRun 346
get_valueRange, YGenericSensor 580
get_voltage, YVSource 1632
get_wakeUpReason, YWakeUpMonitor 1661
get_wakeUpState, YWakeUpMonitor 1662
get_weekDays, YWakeUpSchedule 1700
get_wwwWatchdogDelay, YNetwork 947
get_xValue, YAccelerometer 61
get_xValue, YGyro 632
get_xValue, YMagnetometer 809
get_yValue, YAccelerometer 62
get_yValue, YGyro 633
get_yValue, YMagnetometer 810
get_zValue, YAccelerometer 63
get_zValue, YGyro 634
get_zValue, YMagnetometer 811
GetAPIVersion, YAPI 21
GetTickCount, YAPI 22
Gyroscope 600

H
HandleEvents, YAPI 23
hide, YDisplayLayer 484
hslMove, YColorLed 214
Humidity 676

I
InitAPI, YAPI 24
Interface 36, 78, 120, 158, 197, 226, 266, 305, 376, 420, 467, 499, 524, 551, 600, 651, 676, 715, 742, 784, 832, 872, 913, 993, 1036, 1075, 1123, 1161, 1184, 1223, 1286, 1322, 1361, 1418, 1453, 1494, 1533, 1572, 1611, 1643, 1678, 1715, 1760
Introduction 1
isOnline, YAccelerometer 64
isOnline, YAltitude 104
isOnline, YAnButton 146
isOnline, YCarbonDioxide 183
isOnline, YColorLed 215
isOnline, YCompass 252
isOnline, YCurrent 291
isOnline, YDataLogger 328
isOnline, YDigitallO 402
isOnline, YDisplay 448
isOnline, YDualPower 516
isOnline, YFiles 542
isOnline, YGenericSensor 581
isOnline, YGyro 635
isOnline, YHubPort 668
isOnline, YHumidity 701
isOnline, YLed 732
isOnline, YLightSensor 769
isOnline, YMagnetometer 812
isOnline, YModule 856
isOnline, YMotor 897
isOnline, YNetwork 948
isOnline, YOsControl 985
isOnline, YPower 1021
isOnline, YPressure 1061
isOnline, YPWMInput 1107
isOnline, YPWMOutput 1146
isOnline, YPWMPowerSource 1176
isOnline, YQT 1209
isOnline, YRealTimeClock 1241
isOnline, YRefFrame 1274
isOnline, YRelay 1309
isOnline, YSensor 1347
isOnline, YSerialPort 1385
isOnline, YServo 1439
isOnline, YTemperature 1479
isOnline, YTilt 1519
isOnline, YVoc 1558
isOnline, YVoltage 1597
isOnline, YVSource 1633
isOnline, YWakeUpMonitor 1663
isOnline, YWakeUpSchedule 1701
isOnline, YWatchdog 1742
isOnline, YWireless 1781

J
joinNetwork, YWireless 1782

K
keepALive, YMotor 898

L
LightSensor 742
lineTo, YDisplayLayer 485
load, YAccelerometer 65
load, YAltitude 105
load, YAnButton 147
load, YCarbonDioxide 184
load, YColorLed 216
load, YCompass 253
load, YCurrent 292
load, YDataLogger 329
load, YDigitallO 403
load, YDisplay 449
load, YDualPower 517
load, YFiles 543
load, YGenericSensor 582
load, YGyro 636
load, YHubPort 669
load, YHumidity 702
load, YLed 733
load, YLightSensor 770
load, YMagnetometer 813
load, YModule 857
load, YMotor 899
load, YNetwork 949
load, YOsControl 986
load, YPower 1022
load, YPressure 1062
load, YPwmInput 1108
load, YPwmOutput 1147
load, YPwmPowerSource 1177
load, YQt 1210
load, YRealTimeClock 1242
load, YRefFrame 1275
load, YRelay 1310
load, YSensor 1348
load, YSerialPort 1386
load, YServo 1440
load, YTemperature 1480
load, YTilt 1520
load, YVoc 1559
load, YVoltage 1598
load, YSource 1634
load, YWakeUpMonitor 1664
load, YWakeUpSchedule 1702
load, YWatchdog 1743
load, YWireless 1783
loadCalibrationPoints, YAccelerometer 66
loadCalibrationPoints, YAltitude 106
loadCalibrationPoints, YCarbonDioxide 185
loadCalibrationPoints, YCompass 254
loadCalibrationPoints, YCurrent 293
loadCalibrationPoints, YGenericSensor 583
loadCalibrationPoints, YGyro 637
loadCalibrationPoints, YHumidity 703
loadCalibrationPoints, YLightSensor 771
loadCalibrationPoints, YMagnetometer 814
loadCalibrationPoints, YPower 1023
loadCalibrationPoints, YPressure 1063
loadCalibrationPoints, YPwmInput 1109
loadCalibrationPoints, YQt 1211
loadCalibrationPoints, YSensor 1349
loadCalibrationPoints, YTemperature 1481
loadCalibrationPoints, YTilt 1521
loadCalibrationPoints, YVoc 1560
loadCalibrationPoints, YVoltage 1599
loadMore, YDataSet 359

M
Magnetometer 784
Measured 826
modbusReadBits, YSerialPort 1387
modbusReadInputBits, YSerialPort 1388
modbusReadInputRegisters, YSerialPort 1389
modbusReadRegisters, YSerialPort 1390
modbusWriteAndReadRegisters, YSerialPort 1391
modbusWriteBit, YSerialPort 1392
modbusWriteBits, YSerialPort 1393
modbusWriteRegister, YSerialPort 1394
modbusWriteRegisters, YSerialPort 1395
Module 8, 832
more3DCalibration, YRefFrame 1276
Motor 872
move, YServo 1441
moveTo, YDisplayLayer 486

N
Native 3
Network 913
newSequence, YDisplay 450
nextAccelerometer, YAccelerometer 67
nextAltitude, YAltitude 107
nextAnButton, YAnButton 148
nextCarbonDioxide, YCarbonDioxide 186
nextColorLed, YColorLed 217
nextCompass, YCompass 255
nextCurrent, YCurrent 294
nextDataLogger, YDataLogger 330
nextDigitalIO, YDigitalIO 404
nextDisplay, YDisplay 451
nextDualPower, YDualPower 518
nextFiles, YFiles 544
nextGenericSensor, YGenericSensor 584
nextGyro, YGyro 638
nextHubPort, YHubPort 670
nextHumidity, YHumidity 704
nextLed, YLed 734
nextLightSensor, YLightSensor 772
nextMagnetometer, YMagnetometer 815
nextModule, YModule 858
nextMotor, YMotor 900
nextNetwork, YNetwork 950
nextOsControl, YOsControl 987
nextPower, YPower 1024
nextPressure, YPressure 1064
nextPwmInput, YPwmInput 1110
nextPwmOutput, YPwmOutput 1148
nextPwmPowerSource, YPwmPowerSource 1178
nextQt, YQt 1212
nextRealTimeClock, YRealTimeClock 1243
nextRefFrame, YRefFrame 1277
nextRelay, YRelay 1311
nextSensor, YSensor 1350
nextSerialPort, YSerialPort 1396
nextServo, YServo 1442
nextTemperature, YTemperature 1482
nextTilt, YTilt 1522
nextVoc, YVoc 1561
nextVoltage, YVoltage 1600
nextVSource, YVSource 1635
nextWakeUpMonitor, YWakeUpMonitor 1665
nextWakeUpSchedule, YWakeUpSchedule 1703
nextWatchdog, YWatchdog 1744
nextWireless, YWireless 1784
Object 467

pauseSequence, YDisplay 452
ping, YNetwork 951
playSequence, YDisplay 453
Port 4, 651
Power 499, 993
PreregisterHub, YAPI 25
Pressure 1036
pulse, YDigitalIO 405
pulse, YRelay 1312
pulse, YVSource 1636
pulse, YWatchdog 1745
pulseDurationMove, YPwmOutput 1149
PwmInput 1075
PwmPowerSource 1161

Quaternion 1184
queryLine, YSerialPort 1397
queryMODBUS, YSerialPort 1398

read SEEK, YSerialPort 1403
readHEX, YSerialPort 1399
readLINE, YSerialPort 1400
readMessages, YSerialPort 1401
readSTR, YSerialPort 1402
Real 1223
reboot, YModule 859
Recorded 349
Reference 16, 1250
registerAnglesCallback, YGyro 639
RegisterDeviceArrivalCallback, YAPI 26
RegisterDeviceRemovalCallback, YAPI 27
RegisterHub, YAPI 28
RegisterHubDiscoveryCallback, YAPI 29
registerLogCallback, YModule 860
RegisterLogFunction, YAPI 30
registerQuaternionCallback, YGyro 640
registerTimedReportCallback, YAcelerometer 68
registerTimedReportCallback, YAltitude 108
registerTimedReportCallback, YCarbonDioxide 187
registerTimedReportCallback, YCompass 256
registerTimedReportCallback, YCurrent 295
registerTimedReportCallback, YGenericSensor 585
registerTimedReportCallback, YGyro 641
registerTimedReportCallback, YHumidity 705
registerTimedReportCallback, YLightSensor 773
registerTimedReportCallback, YMagnetometer 816
registerTimedReportCallback, YPower 1025
registerTimedReportCallback, YPressure 1065
registerTimedReportCallback, YPwmInput 1111
registerTimedReportCallback, YQt 1213
registerTimedReportCallback, YSensor 1351
registerTimedReportCallback, YTemperature 1483
registerTimedReportCallback, YTilt 1523
registerTimedReportCallback, YVoc 1562
registerTimedReportCallback, YYVoltage 1601
registerValueCallback, YAccelerometer 69
registerValueCallback, YAltitude 109
registerValueCallback, YANButton 149
registerValueCallback, YCarbonDioxide 188
registerValueCallback, YColorLED 218
registerValueCallback, YCompass 257
registerValueCallback, YCurrent 296
registerValueCallback, YDataLogger 331
registerValueCallback, YDigitalIO 406
registerValueCallback, YDisplay 454
registerValueCallback, YDualPower 519
registerValueCallback, YFiles 545
registerValueCallback, YGenericSensor 586
registerValueCallback, YGyro 642
registerValueCallback, YHubPort 671
registerValueCallback, YHumidity 706
registerValueCallback, YLED 735
registerValueCallback, YLightSensor 774
registerValueCallback, YMagnetometer 817
registerValueCallback, YMotor 901
registerValueCallback, YNetwork 952
registerValueCallback, YOSControl 988
registerValueCallback, YPower 1026
registerValueCallback, YPressure 1066
registerValueCallback, YPwmInput 1112
registerValueCallback, YPwmOutput 1150
registerValueCallback, YRealTimeClock 1244
registerValueCallback, YSerialPort 1404
registerValueCallback, YServo 1443
registerValueCallback, YTemperature 1484
registerValueCallback, YTilt 1524
registerValueCallback, YYVoltage 1602
registerValueCallback, YVSource 1637
registerValueCallback, YWakeUpMonitor 1666
registerValueCallback, YWakeUpSchedule 1704
registerValueCallback, YWatchdog 1746
registerValueCallback, YWireless 1785
Relay 1286
remove, YFiles 546
reset, YDisplayLayer 487
reset, YPower 1027
reset, YSerialPort 1405
resetAll, YDisplay 455
resetCounter, YANButton 150
resetCounter, YPwmInput 1113
resetSleepCountDown, YWakeUpMonitor 1667
resetStatus, YMotor 902
resetWatchdog, YWatchdog 1747
revertFromFlash, YModule 861
rgbMove, YColorLed 219

save3DCalibration, YRefFrame 1279
saveSequence, YDisplay 456
saveToFlash, YModule 862
selectColorPen, YDisplayLayer 488
selectEraser, YDisplayLayer 489
selectFont, YDisplayLayer 490
selectGrayPen, YDisplayLayer 491
Sensor 1322
Sequence 339, 349, 361
SerialPort 1361
Servo 1418

set_adminPassword, YNetwork 953
set_allSettings, YModule 863
set_analogCalibration, YAnButton 151
set_autoStart, YDataLogger 332
set_autoStart, YWakeUpMonitor 1748
set_beacon, YModule 864
set_beaconDriven, YDataLogger 333
set_bearing, YRefFrame 1280
set_bitDirection, YDigitalIO 407
set_bitOpenDrain, YDigitalIO 408
set_bitPolarity, YDigitalIO 409
set_bitState, YDigitalIO 410
set_blinking, YLed 736
set_brakingForce, YMotor 903
set_brightness, YDisplay 457
set_calibrationMax, YAnButton 152
set_calibrationMin, YAnButton 153
set_callbackCredentials, YNetwork 955
set_callbackEncoding, YNetwork 956
set_callbackMaxDelay, YNetwork 957
set_callbackMinDelay, YNetwork 958
set_callbackUrl, YNetwork 959
set_currentValue, YAltitude 110
set_cutOffVoltage, YMotor 904
set_discoverable, YNetwork 960
set_drivingForce, YMotor 905
set_dutyCycle, YPwmOutput 1151
set_dutyCycleAtPowerOn, YPwmOutput 1152
set_enabled, YDisplay 458
set_enabled, YHubPort 672
set_enabled, YPwmOutput 1153
set_enabled, YServo 1444
set_enabledAtPowerOn, YPwmOutput 1154
set_enabledAtPowerOn, YServo 1445
set_failSafeTimeout, YMotor 906
set_frequency, YMotor 907
set_frequency, YPwmOutput 1155
set_highestValue, YAccelerometer 70
set_highestValue, YAltitude 111
set_highestValue, YCarbonDioxide 189
set_highestValue, YCompass 258
set_highestValue, YCurrent 297
set_highestValue, YGenericSensor 587
set_highestValue, YGyro 643
set_highestValue, YHumidity 707
set_highestValue, YLightSensor 775
set_highestValue, YMagnetometer 818
set_highestValue, YPower 1028
set_highestValue, YPressure 1067
set_highValue, YPwmInput 1114
set_highestValue, YYWatchdog 1715
set_highestValue, YSensor 1353
set_highestValue, YTemperature 1485
set_highestValue, YTilt 1525
set_highestValue, YVoltage 1603
set_hours, YWakeUpSchedule 1705
set_hslColor, YColorLed 220
set_logFrequency, YAccelerometer 71
set_logFrequency, YAltitude 112
set_logFrequency, YCarbonDioxide 190
set_logFrequency, YCompass 259
set_logFrequency, YCurrent 298
set_logFrequency, YGenericSensor 588
set_logFrequency, YGyro 644
set_logFrequency, YHumidity 708
set_logFrequency, YLightSensor 776
set_logFrequency, YMagnetometer 819
set_logFrequency, YPower 1029
set_logFrequency, YPressure 1068
set_logFrequency, YPwmInput 1115
set_logFrequency, YQI 1216
set_logFrequency, YSensor 1354
set_logFrequency, YTemperature 1486
set_logFrequency, YTilt 1526
set_logFrequency, YVoltage 1604
set_logicalName, YAccelerometer 72
set_logicalName, YAltitude 113
set_logicalName, YCarbonDioxide 191
set_logicalName, YColorLed 221
set_logicalName, YCompass 260
set_logicalName, YCurrent 299
set_logicalName, YDataLogger 334
set_logicalName, YDigitalIO 411
set_logicalName, YDisplay 459
set_logicalName, YDualPower 520
set_logicalName, YFiles 547
set_logicalName, YGenericSensor 589
set_logicalName, YGyro 645
set_logicalName, YHubPort 673
set_logicalName, YHumidity 709
set_logicalName, YLed 737
set_logicalName, YLightSensor 777
set_logicalName, YMagnetometer 820
set_logicalName, YModule 865
set_logicalName, YMotor 908
set_rgbColor, YColorLed 222
set_rgbColorAtPowerOn, YColorLed 223
set_running, YWatchdog 1753
set_secondaryDNS, YNetwork 963
set_sensitivity, YAnButton 155
set_sensorType, YTemperature 1491
set_serialMode, YSerialPort 1409
set_running, YWatchdog 1753
set_secondaryDNS, YNetwork 963
set_sensitivity, YAnButton 155
set_sensorType, YTemperature 1491
set_serialMode, YSerialPort 1409
set_sleepCountdown, YWakeUpMonitor 1671
set_startupSeq, YMotor 910
set_startupSeq, YDisplay 461
set_state, YRelay 1318
set_state, YWatchdog 1754
set_stateAtPowerOn, YRelay 1319
set_stateAtPowerOn, YWatchdog 1755
set_timeUTC, YDataLogger 336
set_triggerDelay, YWatchdog 1756
set_triggerDuration, YWatchdog 1757
set_unit, YGenericSensor 595
set_unixTime, YRealTimeClock 1246
set_userData, YAccelerometer 76
set_userData, YAltitude 118
set_userData, YAnButton 156
set_userData, YCarbonDioxide 195
set_userData, YColorLed 224
set_userData, YCompass 264
set_userData, YCurrent 303
set_userData, YDataLogger 337
set_userData, YDigitalIO 417
set_userData, YDisplay 462
set_userData, YDualPower 522
set_userData, YFiles 548
set_userData, YGenericSensor 596
set_userData, YGyro 649
set_userData, YHubPort 674
set_userData, YHumidity 713
set_userData, YLed 740
set_userData, YLightSensor 782
set_userData, YMagnetometer 824
set_userData, YModule 867
set_userData, YMotor 911
set_userData, YNetwork 964
set_userData, YOsControl 990
set_userData, YPower 1034
set_userData, YPressure 1073
set_userData, YPwmInput 1121
set_userData, YPwmOutput 1159
set_userData, YPwmPowerSource 1182
set_userData, YQt 1221
set_userData, YRealTimeClock 1247
set_userData, YRefFrame 1283
set_userData, YRelay 1320
set_userData, YSensor 1359
set_userData, YSerialPort 1410
set_userData, YServo 1451
set_userData, YTemperature 1492
set_userData, YTilt 1531
set_userData, YVoc 1570
set_userData, YVoltage 1609
set_userData, YVSource 1639
set_userData, YWakeUpMonitor 1672
set_userData, YWakeUpSchedule 1712
set_userData, YWatchdog 1758
set_userData, YWireless 1787
set_userPassword, YNetwork 965
set_userVar, YModule 868
set_utcOffset, YRealTimeClock 1248
set_valueInterval, YDataRun 347
set_valueRange, YGenericSensor 597
set_voltage, YVSource 1640
set_weekDays, YWakeUpSchedule 1713
set_wwwWatchdogDelay, YNetwork 966
setAntialiasingMode, YDisplayLayer 492
setConsoleBackground, YDisplayLayer 493
setConsoleMargins, YDisplayLayer 494
setConsoleWordWrap, YDisplayLayer 495
setLayerPosition, YDisplayLayer 496
shutdown, YOsControl 991
Sleep, YAPI 31
sleep, YWakeUpMonitor 1673
sleepFor, YWakeUpMonitor 1674
sleepUntil, YWakeUpMonitor 1675
softAPNetwork, YWireless 1788
Source 1611
start3DCalibration, YRefFrame 1284
stopSequence, YDisplay 463
Supply 499
swapLayerContent, YDisplay 464

T
Temperature 1453
Tilt 1494
Time 1223
toggle_bitState, YDigitalIO 418
triggerFirmwareUpdate, YModule 869
TriggerHubDiscovery, YAPI 32

U
Unformatted 361
unhide, YDisplayLayer 497
UnregisterHub, YAPI 33
UpdateDeviceList, YAPI 34
updateFirmware, YModule 870
upload, YDisplay 465
upload, YFiles 549
useDHCP, YNetwork 967
useStaticIP, YNetwork 968

V
Value 826
VirtualHub 3
Voltage 1572, 1611
voltageMove, YVSource 1641
W
wakeUp, YWakeUpMonitor 1676
WakeUpMonitor 1643
WakeUpSchedule 1678
Watchdog 1715
Wireless 1760
writeArray, YSerialPort 1411
writeBin, YSerialPort 1412
writeHex, YSerialPort 1413
writeLine, YSerialPort 1414
writeMODBUS, YSerialPort 1415
writeStr, YSerialPort 1416

Y
YAccelerometer 38-76
YAltitude 80-118
YAnButton 122-156
YAPI 18-34
YCarbonDioxide 160-195
yCheckLogicalName 18
YColorLed 198-224
YCompass 228-264
YCurrent 268-303
YDataLogger 307-337
YDataRun 339-347
YDataSet 350-359
YDataStream 362-374
YDigitalIO 378-418
YDisplay 422-465
YDisplayLayer 468-497
YDualPower 500-522
yEnableUSBHost 19
YFiles 525-549
yFindAccelerometer 38
yFindAltitude 80
yFindAnButton 122
yFindCarbonDioxide 160
yFindColorLed 198
yFindCompass 228
yFindCurrent 268
yFindDataLogger 307
yFindDigitalIO 378
yFindDisplay 422
yFindDualPower 500
yFindFiles 525
yFindGenericSensor 553
yFindGyro 602
yFindHubPort 652
yFindHumidity 678
yFindLed 716
yFindLightSensor 744
yFindMagnetometer 786
yFindModule 834
yFindMotor 874
yFindNetwork 916
yFindOsControl 971
yFindPower 995
yFindPressure 1038
yFindPwmInput 1077
yFindPwmOutput 1125
yFindPwmPowerSource 1162
yFindQt 1186
yFindRealTimeClock 1224
yFindRefFrame 1252
yFindRelay 1288
yFindSensor 1324
yFindSerialPort 1364
yFindServo 1420
yFindTemperature 1455
yFindTilt 1496
yFindVoc 1535
yFindVoltage 1574
yFindVSource 1612
yFindWakeUpMonitor 1645
yFindWakeUpSchedule 1680
yFindWatchdog 1717
yFindWireless 1761
yFirstAccelerometer 39
yFirstAltitude 81
yFirstAnButton 123
yFirstCarbonDioxide 161
yFirstColorLed 199
yFirstCompass 229
yFirstCurrent 269
yFirstDataLogger 308
yFirstDigitalIO 379
yFirstDisplay 423
yFirstDualPower 501
yFirstFiles 526
yFirstGenericSensor 554
yFirstGyro 603
yFirstHubPort 653
yFirstHumidity 679
yFirstLed 717
yFirstLightSensor 745
yFirstMagnetometer 787
yFirstModule 835
yFirstMotor 875
yFirstNetwork 917
yFirstOsControl 972
yFirstPower 996
yFirstPressure 1039
yFirstPwmInput 1078
yFirstPwmOutput 1126
yFirstPwmPowerSource 1163
yFirstQt 1187
yFirstRealTimeClock 1225
yFirstRefFrame 1253
yFirstRelay 1289
yFirstSensor 1325
yFirstSerialPort 1365
yFirstServo 1421
yFirstTemperature 1456
yFirstTilt 1497
yFirstVoc 1536
yFirstVoltage 1575
yFirstVSource 1613