



Java for Android API Reference

Table of contents

Introduction	3
Using the Yocto—Demo with Android	4
Control of the Led function	4
Control of the module part	5
Error handling	7
Reference	8
General functions	8
AnButton function interface	14
CarbonDioxide function interface	23
ColorLed function interface	31
Current function interface	39
DataLogger function interface	48
Formatted data sequence	57
Recorded data sequence	59
External power supply control interface	62
Yocto-hub port interface	69
Humidity function interface	76
Led function interface	84
LightSensor function interface	91
Module control interface	100
Pressure function interface	110
Relay function interface	118
Servo function interface	126
Temperature function interface	133
Voltage function interface	142
Voltage source function interface	150
Index	160

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

[English version of web\FR\doc\doc-android-intro-FR is missing]

2.1. Control of the Led function

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

```
[...]

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

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

[...]
```

Let's look at these lines in more details.

YAPI.RegisterHub

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

YLed.FindLed

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

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

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

isOnline

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

set_power

The `set_power()` function of the object 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-GettingStarted-Yocto-Demo** of the Yoctopuce library.

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

UNABLE TO INCLUDE

<http://172.17.17.77/tu/projects/yoctopoc/public/examples/java/helloworld.java>

2.2. Control of the module part

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

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

public class Demo {

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

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

        try {
            if (args.length > 1) {
                if (args[1].equalsIgnoreCase("ON")) {
                    module.setBeacon(YModule.BEACON_ON);
                } else {
                    module.setBeacon(YModule.BEACON_OFF);
                }
            }
            System.out.println("serial:      " + module.getSerialNumber());
            System.out.println("logical name: " + module.get_logicalName());
            System.out.println("luminosity:   " + module.get_luminosity());
            if (module.getBeacon() == YModule.BEACON_ON) {
                System.out.println("beacon:      ON");
            } else {
                System.out.println("beacon:      OFF");
            }
        }
    }
}
```

```

        System.out.println("upTime:      " + module.getUpTime() / 1000 + " sec"
    );
    System.out.println("USB current:  " + module.getUsbCurrent() + " mA");
    } catch (YAPI_Exception ex) {
        System.out.println(args[1] + " not connected (check identification and
USB cable)");
    }
    YAPI.FreeAPI();
}
}

```

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

Changing the module settings

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

```

import com.yoctopuce.YoctoAPI.*;

public class Demo {

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

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

        YModule m;
        String newname;

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

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

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

            System.out.println("Module: serial= " + m.get_serialNumber());
            System.out.println(" / name= " + m.get_logicalName());
        } catch (YAPI_Exception ex) {
            System.out.println("Module " + args[0] + "not connected (check
identification and USB cable)");
            System.out.println(ex.getMessage());
            System.exit(1);
        }

        YAPI.FreeAPI();
    }
}

```

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

Listing the modules

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

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

public class Demo {

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

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

        YAPI.FreeAPI();
    }
}
```

2.3. Error handling

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

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

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:

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

Global functions

`yCheckLogicalName(name)`

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

`yDisableExceptions()`

Disables the use of exceptions to report runtime errors.

`yEnableExceptions()`

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

`yEnableUSBHost(osContext)`

This function is used only on Android.

`yFreeAPI()`

Frees dynamically allocated memory blocks used by the Yoctopuce library.

`yGetAPIVersion()`

Returns the version identifier for the Yoctopuce library in use.

`yGetTickCount()`

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

`yHandleEvents(errmsg)`

Maintains the device-to-library communication channel.

`yInitAPI(mode, errmsg)`

Initializes the Yoctopuce programming library explicitly.

`yRegisterDeviceArrivalCallback(arrivalCallback)`

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

`yRegisterDeviceRemovalCallback(removalCallback)`

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

yRegisterHub(url, errmsg)

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

yRegisterLogFunction(logfun)

Register a log callback function.

ySetDelegate(object)

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

ySetTimeout(callback, ms_timeout, optional_arguments)

Invoke the specified callback function after a given timeout.

ySleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

yUnregisterHub(url)

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

yUpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName()

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.

Disables the use of exceptions to report runtime errors.

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

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

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

YAPI.EnableUSBHost()

This function is used only on Android.

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

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

Parameters :

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

On failure, throws an exception.

YAPI.FreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

```
void FreeAPI()
```

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

YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

```
String GetAPIVersion()
```

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

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

Returns :

a character string describing the library version.

YAPI.GetTickCount()

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

```
long GetTickCount()
```

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

Returns :

a long integer corresponding to the millisecond counter.

YAPI.HandleEvents()

Maintains the device-to-library communication channel.

```
void 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.

```
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.RegisterDeviceArrivalCallback()

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

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

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

Parameters :

arrivalCallback a procedure taking a `YModule` parameter, or null to unregister a previously registered callback.

YAPI.RegisterDeviceRemovalCallback()

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

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

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

Parameters :

removalCallback a procedure taking a `YModule` parameter, or null to unregister a previously registered callback.

YAPI.RegisterHub()

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

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

When using Yoctopuce modules through the VirtualHub gateway, you should provide as parameter the address of the machine on which the VirtualHub software is running (typically "http://127.0.0.1:4444", which represents the local machine). When you use a language which has direct access to the USB hardware, you can use the pseudo-URL "usb" instead.

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 acces control has been activated on the VirtualHub you want to reach, the URL parameter should look like: `http://username:password@adresse:port`

Parameters :

url a string containing either "usb" 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.RegisterLogFunction()

Register a log callback function.

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

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

Parameters :

logfun a procedure taking a string parameter, or null
to unregister a previously registered callback.

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

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

Parameters :

object an object that must follow the protocol YAPIDelegate, or nil
to unregister a previously registered object.

Invoke the specified callback function after a given timeout.

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

Parameters :

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

Returns :

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

YAPI.Sleep()

Pauses the execution flow for a specified duration.

```
void 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.UnregisterHub()

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

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

Parameters :

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

YAPI.UpdateDeviceList()

Triggers a (re)detection of connected Yoctopuce modules.

```
int UpdateDeviceList()
```

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

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

Parameters :

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

Returns :

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

Triggers a (re)detection of connected Yoctopuce modules.

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

3.2. AnButton function interface

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

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

```
import com.yoctopuce.YoctoAPI.YAnButton;
```

Global functions

yFindAnButton(func)

Retrieves an analog input for a given identifier.

yFirstAnButton()

Starts the enumeration of analog inputs currently accessible.

YAnButton methods

anbutton→describe()

Returns a descriptive text that identifies the function.

anbutton→get_advertisedValue()

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

anbutton→get_analogCalibration()

Tells if a calibration process is currently ongoing.

anbutton→get_calibratedValue()

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

anbutton→get_calibrationMax()

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

anbutton→get_calibrationMin()

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

anbutton→get_errorMessage()

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

anbutton→get_errorType()

Returns the numerical error code of the latest error with this function.

anbutton→get_functionDescriptor()

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

anbutton→get_hardwareId()

Returns the unique hardware identifier of the function.

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

anbutton→get_module_async(callback, context)

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

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 255, included) 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 function is currently reachable, without raising any error.

anbutton→isOnline_async(callback, context)

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

anbutton→load(msValidity)

Preloads the function cache with a specified validity duration.

anbutton→load_async(msValidity, callback, context)

Preloads the function 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→set_analogCalibration(newval)

Starts or stops the calibration process.

anbutton→set_calibrationMax(newval)

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

anbutton→set_calibrationMin(newval)

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

anbutton→set_logicalName(newval)

Changes the logical name of the analog input.

anbutton→set_sensitivity(newval)

Changes the sensibility for the input (between 1 and 255, included) for triggering user callbacks.

anbutton→**set_userdata(data)**

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

YAnButton.FindAnButton()

Retrieves an analog input for a given identifier.

```
YAnButton FindAnButton( String func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the analog input

Returns :

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

YAnButton.FirstAnButton()

Starts the enumeration of analog inputs currently accessible.

```
YAnButton FirstAnButton( )
```

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

Returns :

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

anbutton.describe()

Returns a descriptive text that identifies the function.

```
String describe( )
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

anbutton.get_advertisedValue()

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

```
String get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the analog input (no more than 6 characters)

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

`anbutton.get_analogCalibration()`

Tells if a calibration process is currently ongoing.

```
int get_analogCalibration()
```

Returns :

either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

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

`anbutton.get_calibratedValue()`

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

```
int get_calibratedValue()
```

Returns :

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

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

`anbutton.get_calibrationMax()`

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

```
int get_calibrationMax()
```

Returns :

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

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

`anbutton.get_calibrationMin()`

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

```
int get_calibrationMin()
```

Returns :

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

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

`anbutton.get_errorMessage()`

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

```
String get_errorMessage()
```

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

Returns :

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

`anbutton.get_errorType()`

Returns the numerical error code of the latest error with this function.

```
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 function object

`anbutton.get_anbuttonDescriptor()`

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

`anbutton.get_isPressed()`

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

```
int get_isPressed()
```

Returns :

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

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

`anbutton.get_lastTimePressed()`

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()

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

```
long get_lastTimeReleased()
```

Returns :

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

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

anbutton.get_logicalName()

Returns the logical name of the analog input.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the analog input

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

anbutton.get_module()

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

```
YModule get_module()
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

anbutton.get_rawValue()

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

```
int get_rawValue()
```

Returns :

an integer corresponding to the current measured input value as-is (between 0 and 4095, included)
On failure, throws an exception or returns `Y_RAWVALUE_INVALID`.

`anbutton.get_sensitivity()`

Returns the sensibility for the input (between 1 and 255, included) for triggering user callbacks.

```
int get_sensitivity( )
```

Returns :

an integer corresponding to the sensibility for the input (between 1 and 255, included) for triggering user callbacks

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

`anbutton.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.

`anbutton.isOnline()`

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

```
boolean isOnline( )
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

anbutton.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

anbutton.nextAnButton()

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

```
YAnButton nextAnButton()
```

Returns :

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

anbutton.registerValueCallback()

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.

anbutton.set_analogCalibration()

Starts or stops the calibration process.

```
int set_analogCalibration( int newval)
```

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

Parameters :

newval either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton.set_calibrationMax()

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

```
int set_calibrationMax( int newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton.set_calibrationMin()

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

```
int set_calibrationMin( int newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton.set_logicalName()

Changes the logical name of the analog input.

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton.set_sensitivity()

Changes the sensibility for the input (between 1 and 255, included) for triggering user callbacks.

```
int set_sensitivity( int newval)
```

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 255, included) for triggering user callbacks

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton.set_userData()

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.3. 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:

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

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 descriptive text that identifies the function.

carbondioxide→get_advertisedValue()

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

carbondioxide→get_currentRawValue()

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

carbondioxide→get_currentValue()

Returns the current measured value.

carbondioxide→get_errorMessage()

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

carbondioxide→get_errorType()

Returns the numerical error code of the latest error with this function.

carbondioxide→get_functionDescriptor()

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

carbondioxide→get_hardwareId()

Returns the unique hardware identifier of the function.

carbondioxide→get_highestValue()

Returns the maximal value observed.

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

Returns the minimal value observed.

carbondioxide→get_module()

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

carbondioxide→get_module_async(callback, context)

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_unit()

Returns the measuring unit for the measured value.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

carbondioxide→isOnline_async(callback, context)

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

carbondioxide→load(msValidity)

Preloads the function cache with a specified validity duration.

carbondioxide→load_async(msValidity, callback, context)

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

carbondioxide→nextCarbonDioxide()

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

`carbondioxide→registerValueCallback(callback)`

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

`carbondioxide→set_highestValue(newval)`

Changes the recorded maximal value observed.

`carbondioxide→set_logicalName(newval)`

Changes the logical name of the CO2 sensor.

`carbondioxide→set_lowestValue(newval)`

Changes the recorded minimal value observed.

`carbondioxide→set_userData(data)`

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

`YCarbonDioxide.FindCarbonDioxide()`

Retrieves a CO2 sensor for a given identifier.

```
YCarbonDioxide FindCarbonDioxide( String func)
```

The identifier can be specified using several formats:

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

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

Parameters :

`func` a string that uniquely characterizes the CO2 sensor

Returns :

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

`YCarbonDioxide.FirstCarbonDioxide()`

Starts the enumeration of CO2 sensors currently accessible.

```
YCarbonDioxide FirstCarbonDioxide()
```

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

Returns :

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

`carbondioxide.calibrateFromPoints()`

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

```
int calibrateFromPoints( double rawValues, double refValues)
```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`carbondioxide.describe()`

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

`carbondioxide.get_advertisedValue()`

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

```
String get_advertisedValue()
```

Returns :

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

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

`carbondioxide.get_currentRawValue()`

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

```
double get_currentRawValue()
```

Returns :

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

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

`carbondioxide.get_currentValue()`

Returns the current measured value.

```
double get_currentValue()
```

Returns :

a floating point number corresponding to the current measured value

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

`carbondioxide.get_errorMessage()`

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

```
String get_errorMessage()
```

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

Returns :

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

`carbondioxide.get_errorType()`

Returns the numerical error code of the latest error with this function.

```
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 function object

`carbondioxide.get_carbondioxideDescriptor()`

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

`carbondioxide.get_highestValue()`

Returns the maximal value observed.

```
double get_highestValue()
```

Returns :

a floating point number corresponding to the maximal value observed

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

`carbondioxide.get_logicalName()`

Returns the logical name of the CO2 sensor.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the CO2 sensor

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

carbondioxide.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed

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

carbondioxide.get_module()

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

```
YModule get_module( )
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

carbondioxide.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

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

Returns :

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

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

carbondioxide.get_unit()

Returns the measuring unit for the measured value.

```
String get_unit()
```

Returns :

a string corresponding to the measuring unit for the measured value

On failure, throws an exception or returns Y_UNIT_INVALID.

carbondioxide.get_userData()

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

```
Object get_userData()
```

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

Returns :

the object stored previously by the caller.

carbondioxide.isOnline()

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

```
boolean isOnline()
```

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

Returns :

true if the function can be reached, and false otherwise

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

carbondioxide.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

carbondioxide.nextCarbonDioxide()

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

```
YCarbonDioxide nextCarbonDioxide()
```

Returns :

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

carbondioxide.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.

carbondioxide.set_highestValue()

Changes the recorded maximal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide.set_logicalName()

Changes the logical name of the CO2 sensor.

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide.set_lowestValue()

Changes the recorded minimal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide.set_userData()

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.4. 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:

```
import com.yoctopuce.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 descriptive text that identifies the function.

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 this function.

colorled→get_errorType()

Returns the numerical error code of the latest error with this function.

colorled→get_functionDescriptor()

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

colorled→get_hardwareId()

Returns the unique hardware identifier of the function.

colorled→get_hslColor()

Returns the current HSL color of the led.

colorled→get_logicalName()

Returns the logical name of the RGB led.

colorled→get_module()

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

colorled→get_module_async(callback, context)

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

colorled→get_rgbColor()

Returns the current RGB color of the led.

colorled→get_rgbColorAtPowerOn()

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

colorled→get_userData()

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

colorled→hslMove(hsl_target, ms_duration)

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

colorled→isOnline()

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

colorled→isOnline_async(callback, context)

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

colorled→load(msValidity)

Preloads the function cache with a specified validity duration.

colorled→load_async(msValidity, callback, context)

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

colorled→nextColorLed()

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

colorled→registerValueCallback(callback)

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

colorled→rgbMove(rgb_target, ms_duration)

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

colorled→set_hslColor(newval)

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

colorled→set_logicalName(newval)

Changes the logical name of the RGB led.

colorled→set_rgbColor(newval)

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

colorled→set_rgbColorAtPowerOn(newval)

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

colorled→set_userData(data)

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

YColorLed.FindColorLed()

Retrieves an RGB led for a given identifier.

```
YColorLed FindColorLed( String func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the RGB led

Returns :

a `YColorLed` object allowing you to drive the RGB led.

YColorLed.FirstColorLed()

Starts the enumeration of RGB leds currently accessible.

```
YColorLed FirstColorLed( )
```

Use the method `YColorLed.nextColorLed()` to iterate on next RGB leds.

Returns :

a pointer to a `YColorLed` object, corresponding to the first RGB led currently online, or a `null` pointer if there are none.

colorled.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

colorled.get_advertisedValue()

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

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the RGB led (no more than 6 characters)

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

colorled.get_errorMessage()

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

```
String get_errorMessage()
```

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

Returns :

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

colorled.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

colorled.get_colorledDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

colorled.get_hslColor()

Returns the current HSL color of the led.

```
int get_hslColor()
```

Returns :

an integer corresponding to the current HSL color of the led

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

colorled.get_logicalName()

Returns the logical name of the RGB led.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the RGB led

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

colorled.get_module()

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

```
YModule get_module()
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

colorled.get_rgbColor()

Returns the current RGB color of the led.

```
int get_rgbColor( )
```

Returns :

an integer corresponding to the current RGB color of the led

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

colorled.get_rgbColorAtPowerOn()

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

```
int get_rgbColorAtPowerOn( )
```

Returns :

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

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

colorled.get_userData()

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()

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

```
int hslMove( int hsl_target, int ms_duration)
```

Parameters :

hsl_target desired HSL color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

colorled.isOnline()

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

```
boolean isOnline( )
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

colored.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

colored.nextColorLed()

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

```
YColorLed nextColorLed( )
```

Returns :

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

colorled.registerValueCallback()

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.

colorled.rgbMove()

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

```
int rgbMove( int rgb_target, int ms_duration)
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled.set_hslColor()

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

```
int set_hslColor( int newval)
```

Encoding is done as follows: 0xHHSSL.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled.set_logicalName()

Changes the logical name of the RGB led.

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

colorled.set_rgbColor()

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

```
int set_rgbColor( int newval)
```

Encoding is done as follows: 0xRRGGBB.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

colorled.set_rgbColorAtPowerOn()

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

```
int set_rgbColorAtPowerOn( int newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

colorled.set_userData()

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.5. 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:

```
import com.yoctopuce.YoctoAPI.YCurrent;
```

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 descriptive text that identifies the function.

current→get_advertisedValue()

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

current→get_currentRawValue()

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

current→get_currentValue()

Returns the current measured value.

current→get_errorMessage()

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

current→get_errorType()

Returns the numerical error code of the latest error with this function.

current→get_functionDescriptor()

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

current→get_hardwareId()

Returns the unique hardware identifier of the function.

current→get_highestValue()

Returns the maximal value observed.

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

Returns the minimal value observed.

current→get_module()

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

current→get_module_async(callback, context)

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

current→get_resolution()

Returns the resolution of the measured values.

current→get_unit()

Returns the measuring unit for the measured value.

current→get_userData()

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

current→isOnline()

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

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

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

current→**load**(msValidity)

Preloads the function cache with a specified validity duration.

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

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

current→**nextCurrent**()

Continues the enumeration of current sensors started using `yFirstCurrent()`.

current→**registerValueCallback**(callback)

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

current→**set_highestValue**(newval)

Changes the recorded maximal value observed.

current→**set_logicalName**(newval)

Changes the logical name of the current sensor.

current→**set_lowestValue**(newval)

Changes the recorded minimal value observed.

current→**set_userData**(data)

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

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()

Starts the enumeration of current sensors currently accessible.

```
YCurrent FirstCurrent( )
```

Use the method `YCurrent.nextCurrent()` to iterate on next current sensors.

Returns :

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

current.calibrateFromPoints()

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

```
int calibrateFromPoints( double rawValues, double refValues)
```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

current.get_advertisedValue()

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

```
String get_advertisedValue()
```

Returns :

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

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

current.get_currentRawValue()

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

```
double get_currentRawValue()
```

Returns :

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

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

current.get_currentValue()

Returns the current measured value.

```
double get_currentValue()
```

Returns :

a floating point number corresponding to the current measured value

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

current.get_errorMessage()

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

```
String get_errorMessage()
```

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

Returns :

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

current.getErrorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

current.getCurrentDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

current.get_highestValue()

Returns the maximal value observed.

```
double get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed

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

current.get_logicalName()

Returns the logical name of the current sensor.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the current sensor

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

current.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed

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

current.get_module()

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

```
YModule get_module( )
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

current.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

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

Returns :

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

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

current.get_unit()

Returns the measuring unit for the measured value.

```
String get_unit( )
```

Returns :

a string corresponding to the measuring unit for the measured value

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

current.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.

current.isOnline()

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

```
boolean isOnline( )
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

current.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

current.nextCurrent()

Continues the enumeration of current sensors started using yFirstCurrent().

```
YCurrent nextCurrent()
```

Returns :

a pointer to a YCurrent object, corresponding to a current sensor currently online, or a null pointer if there are no more current sensors to enumerate.

current.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.

current.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.

current.set_logicalName()

Changes the logical name of the current sensor.

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

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

Parameters :

newval a string corresponding to the logical name of the current sensor

Returns :

YAPI_SUCCESS if the call succeeds.

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

current.set_lowestValue()

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.

current.set_userData()

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

```
void set_userData( Object data)
```

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

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

3.6. 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 Yoctopuce application programming interface includes functions to control how this internal data logger works. Because the sensors do not include a battery, they do not have an absolute time reference. Therefore, measures are simply indexed by the absolute run number and time relative to the start of the run. Every new power up starts a new run. It is however possible to setup an absolute UTC time by software at a given time, so that the data logger keeps track of it until it is next powered off.

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

```
import com.yoctopuce.YoctoAPI.YDataLogger;
```

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 descriptive text that identifies the function.

datalogger→forgetAllDataStreams()

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

datalogger→get_advertisedValue()

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

datalogger→get_autoStart()

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

datalogger→get_currentRunIndex()

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

datalogger→get_dataRun(runIdx)

Returns a data run object holding all measured data for a given period during which the module was turned on (a run).

datalogger→get_dataStreams(v)

Builds a list of all data streams hold by the data logger.

datalogger→get_errorMessage()

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

datalogger→get_errorType()

Returns the numerical error code of the latest error with this function.

datalogger→get_functionDescriptor()

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

datalogger→get_hardwareId()

Returns the unique hardware identifier of the function.

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_measureNames()

Returns the names of the measures recorded by the data logger.

datalogger→get_module()

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

datalogger→get_module_async(callback, context)

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

datalogger→get_oldestRunIndex()

Returns the index of the oldest run for which the non-volatile memory still holds recorded data.

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

datalogger→get_userData()

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

datalogger→isOnline()

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

datalogger→isOnline_async(callback, context)

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

datalogger→load(msValidity)

Preloads the function cache with a specified validity duration.

datalogger→load_async(msValidity, callback, context)

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

datalogger→nextDataLogger()

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

datalogger→registerValueCallback(callback)

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

datalogger→set_autoStart(newval)

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

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(data)

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

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()`

Starts the enumeration of data loggers currently accessible.

```
YDataLogger FirstDataLogger()
```

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

Returns :

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

`datalogger.describe()`

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

`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.

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

Returns :

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

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

`datalogger.get_autoStart()`

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

```
int get_autoStart()
```

Returns :

either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

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

`datalogger.get_currentRunIndex()`

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

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

`datalogger.get_dataRun()`

Returns a data run object holding all measured data for a given period during which the module was turned on (a run).

```
YDataRun get_dataRun(int runIdx)
```

This object can then be used to retrieve measures (min, average and max) at a desired data rate.

Parameters :

`runIdx` the index of the desired run

Returns :

an `YDataRun` object

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

`datalogger.get_dataStreams()`

Builds a list of all data streams hold by the data logger.

```
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.

Parameters :

`v` an array of `YDataStream` objects to be filled in

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`datalogger.get_errorMessage()`

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

```
String get_errorMessage()
```

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

Returns :

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

`datalogger.get_errorType()`

Returns the numerical error code of the latest error with this function.

```
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 function object

`datalogger.get_dataloggerDescriptor()`

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

`datalogger.get_logicalName()`

Returns the logical name of the data logger.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the data logger

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

Returns the names of the measures recorded by the data logger.

In most cases, 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.

`datalogger.get_module()`

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

```
YModule get_module( )
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

`datalogger.get_oldestRunIndex()`

Returns the index of the oldest run for which the non-volatile memory still holds recorded data.

```
int get_oldestRunIndex( )
```

Returns :

an integer corresponding to the index of the oldest run for which the non-volatile memory still holds recorded data

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

`datalogger.get_recording()`

Returns the current activation state of the data logger.

```
int get_recording( )
```

Returns :

either `Y_RECORDING_OFF` or `Y_RECORDING_ON`, according to the current activation state of the data logger

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

`datalogger.get_timeUTC()`

Returns the Unix timestamp for current UTC time, if known.

```
int get_timeUTC( )
```

Returns :

an integer corresponding to the Unix timestamp for current UTC time, if known

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

`datalogger.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.

`datalogger.isOnline()`

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

```
boolean isOnline( )
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

`datalogger.load()`

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

`datalogger.nextDataLogger()`

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

```
YDataLogger nextDataLogger()
```

Returns :

a pointer to a YDataLogger object, corresponding to a data logger currently online, or a null pointer if there are no more data loggers to enumerate.

`datalogger.registerValueCallback()`

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.

`datalogger.set_autoStart()`

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

```
int set_autoStart( int newval)
```

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

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`datalogger.set_logicalName()`

Changes the logical name of the data logger.

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

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

Parameters :

newval a string corresponding to the logical name of the data logger

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`datalogger.set_recording()`

Changes the activation state of the data logger to start/stop recording data.

```
int set_recording( int newval)
```

Parameters :

newval either `Y_RECORDING_OFF` or `Y_RECORDING_ON`, according to the activation state of the data logger to start/stop recording data

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`datalogger.set_timeUTC()`

Changes the current UTC time reference used for recorded data.

```
int set_timeUTC( int newval)
```

Parameters :

newval an integer corresponding to the current UTC time reference used for recorded data

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`datalogger.set_userData()`

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.7. 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:

```
import com.yoctopuce.YoctoAPI.YDataLogger;
```

YDataRun methods

datarun→get_averageValue(measureName, pos)

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

datarun→get_duration()

Returns the duration (in seconds) of the data run.

datarun→get_maxValue(measureName, pos)

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

datarun→get_measureNames()

Returns the names of the measures recorded by the data logger.

datarun→get_minValue(measureName, pos)

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

datarun→get_startTimeUTC()

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

datarun→get_valueCount()

Returns the number of values accessible in this run, given the selected data samples interval.

datarun→get_valueInterval()

Returns the number of seconds covered by each value in this run.

datarun→set_valueInterval(valueInterval)

Changes the number of seconds covered by each value in this run.

datarun.get_averageValue()

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

```
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()`

Returns the duration (in seconds) of the data run.

```
long get_duration()
```

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

Returns :

an unsigned number corresponding to the number of seconds between the beginning of the run (when the module was powered up) and the last recorded measure.

`datarun.get_maxValue()`

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

```
double get_maxValue(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 maximal value)

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

`datarun.get_measureNames()`

Returns the names of the measures recorded by the data logger.

```
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.

`datarun.get_minValue()`

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

```
double get_minValue(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 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).

`datarun.get_valueCount()`

Returns the number of values accessible in this run, given the selected data samples interval.

```
int get_valueCount()
```

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

Returns :

an unsigned number corresponding to the run duration divided by the samples interval.

`datarun.get_valueInterval()`

Returns the number of seconds covered by each value in this run.

```
int get_valueInterval()
```

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Returns :

an unsigned number corresponding to a number of seconds covered by each data sample in the Run.

`datarun.set_valueInterval()`

Changes the number of seconds covered by each value in this run.

```
void set_valueInterval( int valueInterval)
```

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Parameters :

`valueInterval` an integer number of seconds.

Returns :

nothing

3.8. Recorded data sequence

DataStream objects represent a recorded measure sequence. They are returned by the data logger present on Yoctopuce sensors.

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

```
import com.yoctopuce.YoctoAPI.YDataLogger;
```

YDataStream methods

`datastream→get_columnCount()`

Returns the number of data columns present in this stream.

`datastream→get_columnNames()`

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

`datastream→get_data(row, col)`

Returns a single measure from the data stream, specified by its row and column index.

datastream→get_dataRows()

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

datastream→get_dataSamplesInterval()

Returns the number of seconds elapsed between two consecutive rows of this data stream.

datastream→get_rowCount()

Returns the number of data rows present in this stream.

datastream→get_runIndex()

Returns the run index of the data stream.

datastream→get_startTime()

Returns the start time of the data stream, relative to the beginning of the run.

datastream→get_startTimeUTC()

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

datastream.get_columnCount()

Returns the number of data columns present in this stream.

```
int get_columnCount()
```

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

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

Returns :

an unsigned number corresponding to the number of rows. On failure, throws an exception or returns zero.

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

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For archived streams created by summarizing a high-resolution data stream, there can be a suffix appended to the sensor identifier, such as `_min` for the minimum value, `_avg` for the average value and `_max` for the maximal value.

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

Returns :

a list containing as many strings as there are columns in the data stream. On failure, throws an exception or returns an empty array.

datastream.get_data()

Returns a single measure from the data stream, specified by its row and column index.

```
double get_data(int row, int col)
```

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

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

Parameters :

row row index

col column index

Returns :

a floating-point number On failure, throws an exception or returns Y_DATA_INVALID.

datastream.get_dataRows()

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

```
ArrayList<ArrayList<Double>> get_dataRows()
```

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

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

Returns :

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers. On failure, throws an exception or returns an empty array.

datastream.get_dataSamplesInterval()

Returns the number of seconds elapsed between two consecutive rows of this data stream.

```
int get_dataSamplesInterval()
```

By default, the data logger records one row per second, but there might be alternative streams at lower resolution created by summarizing the original stream for archiving purposes.

This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

Returns :

an unsigned number corresponding to a number of seconds.

datastream.get_rowCount()

Returns the number of data rows present in this stream.

```
int get_rowCount()
```

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

Returns :

an unsigned number corresponding to the number of rows. On failure, throws an exception or returns zero.

datastream.get_runIndex()

Returns the run index of the data stream.

```
int get_runIndex()
```

A run can be made of multiple datastreams, for different time intervals. This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

Returns :

an unsigned number corresponding to the run index.

datastream.get_startTime()

Returns the start time of the data stream, relative to the beginning of the run.

```
int get_startTime( )
```

If you need an absolute time, use `get_startTimeUTC()`.

This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

Returns :

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

datastream.get_startTimeUTC()

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

```
long get_startTimeUTC( )
```

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

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).

3.9. 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:

```
import com.yoctopuce.YoctoAPI.YDualPower;
```

Global functions

yFindDualPower(func)

Retrieves a dual power control for a given identifier.

yFirstDualPower()

Starts the enumeration of dual power controls currently accessible.

YDualPower methods

dualpower→describe()

Returns a descriptive text that identifies the function.

dualpower→get_advertisedValue()

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

dualpower→get_errorMessage()

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

dualpower→get_errorType()

Returns the numerical error code of the latest error with this function.

dualpower→get_extVoltage()

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

dualpower→get_functionDescriptor()

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

dualpower→get_hardwareId()

Returns the unique hardware identifier of the function.
dualpower→get_logicalName() Returns the logical name of the power control.
dualpower→get_module() Get the YModule object for the device on which the function is located.
dualpower→get_module_async(callback, context) Get the YModule object for the device on which the function is located (asynchronous version).
dualpower→get_powerControl() Returns the selected power source for module functions that require lots of current.
dualpower→get_powerState() Returns the current power source for module functions that require lots of current.
dualpower→get_userData() Returns the value of the userData attribute, as previously stored using method set_userData.
dualpower→isOnline() Checks if the function is currently reachable, without raising any error.
dualpower→isOnline_async(callback, context) Checks if the function is currently reachable, without raising any error (asynchronous version).
dualpower→load(msValidity) Preloads the function cache with a specified validity duration.
dualpower→load_async(msValidity, callback, context) Preloads the function cache with a specified validity duration (asynchronous version).
dualpower→nextDualPower() Continues the enumeration of dual power controls started using yFirstDualPower().
dualpower→registerValueCallback(callback) Registers the callback function that is invoked on every change of advertised value.
dualpower→set_logicalName(newval) Changes the logical name of the power control.
dualpower→set_powerControl(newval) Changes the selected power source for module functions that require lots of current.
dualpower→set_userData(data) Stores a user context provided as argument in the userData attribute of the function.

YDualPower.FindDualPower()

Retrieves a dual power control for a given identifier.

```
YDualPower FindDualPower( String func)
```

The identifier can be specified using several formats:

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

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method YDualPower.isOnline() to test if

the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

`YDualPower.FirstDualPower()`

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 descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

`dualpower.get_advertisedValue()`

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

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the power control (no more than 6 characters)

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

`dualpower.get_errorMessage()`

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

```
String get_errorMessage()
```

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

Returns :

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

`dualpower.get_errorType()`

Returns the numerical error code of the latest error with this function.

```
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 function object

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_dualpowerDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

dualpower.get_logicalName()

Returns the logical name of the power control.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the power control

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

dualpower.get_module()

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

```
YModule get_module()
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

`dualpower.get_powerControl()`

Returns the selected power source for module functions that require lots of current.

`int get_powerControl()`

Returns :

a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

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

`dualpower.get_powerState()`

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()`

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()`

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

```
boolean isOnline( )
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

dualpower.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :
nothing : the result is provided to the callback.

dualpower.nextDualPower()

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

```
YDualPower nextDualPower()
```

Returns :
a pointer to a `YDualPower` object, corresponding to a dual power control currently online, or a `null` pointer if there are no more dual power controls to enumerate.

dualpower.registerValueCallback()

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.

dualpower.set_logicalName()

Changes the logical name of the power control.

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

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

Parameters :
newval a string corresponding to the logical name of the power control

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

dualpower.set_powerControl()

Changes the selected power source for module functions that require lots of current.

```
int set_powerControl(int newval)
```

Parameters :
newval a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

dualpower.set_userdata()

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

3.10. Yocto-hub port interface

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

```
import com.yoctopuce.YoctoAPI.YHubPort;
```

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 descriptive text that identifies the function.

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 this function.

hubport→get_errorType()

Returns the numerical error code of the latest error with this function.

hubport→get_functionDescriptor()

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

hubport→get_hardwareId()

Returns the unique hardware identifier of the function.

hubport→get_logicalName()

Returns the logical name of the Yocto-hub port, which is always the serial number of the connected module.

hubport→get_module()

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

hubport→get_module_async(callback, context)

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

hubport→get_portState()

Returns the current state of the Yocto-hub port.
hubport→get_userdata() Returns the value of the userData attribute, as previously stored using method <code>set_userdata</code> .
hubport→isOnline() Checks if the function is currently reachable, without raising any error.
hubport→isOnline_async(callback, context) Checks if the function is currently reachable, without raising any error (asynchronous version).
hubport→load(msValidity) Preloads the function cache with a specified validity duration.
hubport→load_async(msValidity, callback, context) Preloads the function cache with a specified validity duration (asynchronous version).
hubport→nextHubPort() Continues the enumeration of Yocto-hub ports started using <code>yFirstHubPort()</code> .
hubport→registerValueCallback(callback) Registers the callback function that is invoked on every change of advertised value.
hubport→set_enabled(newval) Changes the activation of the Yocto-hub port.
hubport→set_logicalName(newval) It is not possible to configure the logical name of a Yocto-hub port.
hubport→set_userdata(data) Stores a user context provided as argument in the userData attribute of the function.

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()

Starts the enumeration of Yocto-hub ports currently accessible.

```
YHubPort FirstHubPort()
```

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

Returns :

a pointer to a `YHubPort` object, corresponding to the first Yocto-hub port currently online, or a null pointer if there are none.

hubport.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

hubport.get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the Yocto-hub port (no more than 6 characters)

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

hubport.get_baudRate()

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 hit.

Returns :

an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

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

hubport.get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

```
int get_enabled()
```

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the Yocto-hub port is powered, false otherwise

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

hubport.get_errorMessage()

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

```
String get_errorMessage( )
```

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

Returns :

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

hubport.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

hubport.get_hubportDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function. On failure, throws an exception or returns Y_HARDWAREID_INVALID.

hubport.get_logicalName()

Returns the logical name of the Yocto-hub port, which is always the serial number of the connected module.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the Yocto-hub port, which is always the serial number of the connected module

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

hubport.get_module()

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

```
YModule get_module( )
```

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

Returns :
an instance of `YModule`

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

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

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

Returns :
nothing : the result is provided to the callback.

hubport.get_portState()

Returns the current state of the Yocto-hub port.

```
int get_portState()
```

Returns :
a value among `Y_PORTSTATE_OFF`, `Y_PORTSTATE_ON` and `Y_PORTSTATE_RUN` corresponding to the current state of the Yocto-hub port

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

hubport.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.

hubport.isOnline()

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

```
boolean isOnline()
```

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

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

hubport.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

hubport.nextHubPort()

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

```
YHubPort nextHubPort( )
```

Returns :

a pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a `null` pointer if there are no more Yocto-hub ports to enumerate.

hubport.registerValueCallback()

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.

hubport.set_enabled()

Changes the activation of the Yocto-hub port.

```
int set_enabled( int newval)
```

If the port is enabled, the * connected module will be powered. Otherwise, port power will be shut down.

Parameters :

newval either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the activation of the Yocto-hub port

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

hubport.set_logicalName()

It is not possible to configure the logical name of a Yocto-hub port.

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

The logical name is automatically set to the serial number of the connected module.

Parameters :

newval a string

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

hubport.set_userData()

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.11. 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:

```
import com.yoctopuce.YoctoAPI.YHumidity;
```

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 descriptive text that identifies the function.

humidity→get_advertisedValue()

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

humidity→get_currentRawValue()

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

humidity→get_currentValue()

Returns the current measured value.

humidity→get_errorMessage()

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

humidity→get_errorType()

Returns the numerical error code of the latest error with this function.

humidity→get_functionDescriptor()

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

humidity→get_hardwareId()

Returns the unique hardware identifier of the function.

humidity→get_highestValue()

Returns the maximal value observed.

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

Returns the minimal value observed.

humidity→get_module()

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

humidity→get_module_async(callback, context)

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

humidity→get_resolution()

Returns the resolution of the measured values.
humidity→get_unit() Returns the measuring unit for the measured value.
humidity→get_userData() Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
humidity→isOnline() Checks if the function is currently reachable, without raising any error.
humidity→isOnline_async(callback, context) Checks if the function is currently reachable, without raising any error (asynchronous version).
humidity→load(msValidity) Preloads the function cache with a specified validity duration.
humidity→load_async(msValidity, callback, context) Preloads the function cache with a specified validity duration (asynchronous version).
humidity→nextHumidity() Continues the enumeration of humidity sensors started using <code>yFirstHumidity()</code> .
humidity→registerValueCallback(callback) Registers the callback function that is invoked on every change of advertised value.
humidity→set_highestValue(newval) Changes the recorded maximal value observed.
humidity→set_logicalName(newval) Changes the logical name of the humidity sensor.
humidity→set_lowestValue(newval) Changes the recorded minimal value observed.
humidity→set_userData(data) Stores a user context provided as argument in the userData attribute of the function.

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()

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()

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

```
int calibrateFromPoints(double rawValues, double refValues)
```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

humidity.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

humidity.get_advertisedValue()

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

```
String get_advertisedValue()
```

Returns :

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

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

humidity.get_currentRawValue()

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

```
double get_currentRawValue()
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

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

humidity.get_currentValue()

Returns the current measured value.

```
double get_currentValue()
```

Returns :

a floating point number corresponding to the current measured value

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

humidity.get_errorMessage()

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

```
String get_errorMessage()
```

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

Returns :

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

humidity.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

humidity.get_humidityDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

humidity.get_highestValue()

Returns the maximal value observed.

```
double get_highestValue()
```

Returns :

a floating point number corresponding to the maximal value observed

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

humidity.get_logicalName()

Returns the logical name of the humidity sensor.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the humidity sensor

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

humidity.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue()
```

Returns :

a floating point number corresponding to the minimal value observed

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

humidity.get_module()

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

```
YModule get_module()
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

humidity.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution()
```

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

Returns :

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

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

humidity.get_unit()

Returns the measuring unit for the measured value.

```
String get_unit()
```

Returns :

a string corresponding to the measuring unit for the measured value

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

humidity.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.

humidity.isOnline()

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

```
boolean isOnline()
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

humidity.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

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.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.

`humidity.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.

`humidity.set_logicalName()`

Changes the logical name of the humidity sensor.

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

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

Parameters :

newval a string corresponding to the logical name of the humidity sensor

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`humidity.set_lowestValue()`

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.

humidity.set_userdata()

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

```
void set_userdata( Object data)
```

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

Parameters :

data any kind of object to be stored

3.12. 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:

```
import com.yoctopuce.YoctoAPI.YLed;
```

Global functions

yFindLed(func)

Retrieves a led for a given identifier.

yFirstLed()

Starts the enumeration of leds currently accessible.

YLed methods

led→describe()

Returns a descriptive text that identifies the function.

led→get_advertisedValue()

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

led→get_blinking()

Returns the current led signaling mode.

led→get_errorMessage()

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

led→get_errorType()

Returns the numerical error code of the latest error with this function.

led→get_functionDescriptor()

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

led→get_hardwareId()

Returns the unique hardware identifier of the function.

led→get_logicalName()

Returns the logical name of the led.

led→get_luminosity()

Returns the current led intensity (in per cent).

led→get_module()

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

led→get_module_async(callback, context)

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

led→get_power()	Returns the current led state.
led→get_userData()	Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
led→isOnline()	Checks if the function is currently reachable, without raising any error.
led→isOnline_async(callback, context)	Checks if the function is currently reachable, without raising any error (asynchronous version).
led→load(msValidity)	Preloads the function cache with a specified validity duration.
led→load_async(msValidity, callback, context)	Preloads the function cache with a specified validity duration (asynchronous version).
led→nextLed()	Continues the enumeration of leds started using <code>yFirstLed()</code> .
led→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
led→set_blinking(newval)	Changes the current led signaling mode.
led→set_logicalName(newval)	Changes the logical name of the led.
led→set_luminosity(newval)	Changes the current led intensity (in per cent).
led→set_power(newval)	Changes the state of the led.
led→set_userData(data)	Stores a user context provided as argument in the userData attribute of the function.

YLed.FindLed()

Retrieves a led for a given identifier.

```
YLed FindLed( String func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the led

Returns :

a `YLed` object allowing you to drive the led.

YLed.FirstLed()

Starts the enumeration of leds currently accessible.

```
YLed FirstLed()
```

Use the method `YLed.nextLed()` to iterate on next leds.

Returns :

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

led.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

led.get_advertisedValue()

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

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the led (no more than 6 characters)

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

led.get_blinking()

Returns the current led signaling mode.

```
int get_blinking()
```

Returns :

a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

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

led.get_errorMessage()

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

```
String get_errorMessage()
```

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

Returns :

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

led.getErrorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

led.get_ledDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

led.get_logicalName()

Returns the logical name of the led.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the led

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

led.get_luminosity()

Returns the current led intensity (in per cent).

```
int get_luminosity( )
```

Returns :

an integer corresponding to the current led intensity (in per cent)

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

led.get_module()

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

```
YModule get_module( )
```

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

Returns :
an instance of `YModule`

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

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

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

Returns :
nothing : the result is provided to the callback.

led.get_power()
Returns the current led state.

```
int get_power()
```

Returns :
either `Y_POWER_OFF` or `Y_POWER_ON`, according to the current led state
On failure, throws an exception or returns `Y_POWER_INVALID`.

led.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.

led.isOnline()
Checks if the function is currently reachable, without raising any error.

```
boolean isOnline()
```

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

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

led.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

led.nextLed()

Continues the enumeration of leds started using `yFirstLed()`.

```
YLed nextLed( )
```

Returns :

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

`led.registerValueCallback()`

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.

`led.set_blinking()`

Changes the current led signaling mode.

```
int set_blinking( int newval)
```

Parameters :

newval a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`led.set_logicalName()`

Changes the logical name of the led.

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

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

Parameters :

newval a string corresponding to the logical name of the led

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`led.set_luminosity()`

Changes the current led intensity (in per cent).

```
int set_luminosity( int newval)
```

Parameters :

newval an integer corresponding to the current led intensity (in per cent)

Returns :

YAPI_SUCCESS if the call succeeds.

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

led.set_power()

Changes the state of the led.

```
int set_power( int newval)
```

Parameters :

newval either Y_POWER_OFF or Y_POWER_ON, according to the state of the led

Returns :

YAPI_SUCCESS if the call succeeds.

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

led.set_userdata()

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

3.13. 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:

```
import com.yoctopuce.YoctoAPI.YLightSensor;
```

Global functions

yFindLightSensor(func)

Retrieves a light sensor for a given identifier.

yFirstLightSensor()

Starts the enumeration of light sensors currently accessible.

YLightSensor methods

lightsensor→calibrate(calibratedVal)

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

lightsensor→calibrateFromPoints(rawValues, refValues)

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

lightsensor→describe()

Returns a descriptive text that identifies the function.

lightsensor→get_advertisedValue()

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

lightsensor→get_currentRawValue()

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

lightsensor→get_currentValue()	Returns the current measured value.
lightsensor→get_errorMessage()	Returns the error message of the latest error with this function.
lightsensor→get_errorType()	Returns the numerical error code of the latest error with this function.
lightsensor→get_functionDescriptor()	Returns a unique identifier of type YFUN_DESCR corresponding to the function.
lightsensor→get_hardwareId()	Returns the unique hardware identifier of the function.
lightsensor→get_highestValue()	Returns the maximal value observed.
lightsensor→get_logicalName()	Returns the logical name of the light sensor.
lightsensor→get_lowestValue()	Returns the minimal value observed.
lightsensor→get_module()	Get the YModule object for the device on which the function is located.
lightsensor→get_module_async(callback, context)	Get the YModule object for the device on which the function is located (asynchronous version).
lightsensor→get_resolution()	Returns the resolution of the measured values.
lightsensor→get_unit()	Returns the measuring unit for the measured value.
lightsensor→get_userData()	Returns the value of the userData attribute, as previously stored using method set_userData.
lightsensor→isOnline()	Checks if the function is currently reachable, without raising any error.
lightsensor→isOnline_async(callback, context)	Checks if the function is currently reachable, without raising any error (asynchronous version).
lightsensor→load(msValidity)	Preloads the function cache with a specified validity duration.
lightsensor→load_async(msValidity, callback, context)	Preloads the function cache with a specified validity duration (asynchronous version).
lightsensor→nextLightSensor()	Continues the enumeration of light sensors started using yFirstLightSensor().
lightsensor→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
lightsensor→set_highestValue(newval)	Changes the recorded maximal value observed.
lightsensor→set_logicalName(newval)	Changes the logical name of the light sensor.
lightsensor→set_lowestValue(newval)	

Changes the recorded minimal value observed.

lightsensor→**set_userdata(data)**

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

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()

Starts the enumeration of light sensors currently accessible.

```
YLightSensor FirstLightSensor()
```

Use the method `YLightSensor.nextLightSensor()` to iterate on next light sensors.

Returns :

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

lightsensor.calibrate()

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```
int calibrate( double calibratedVal)
```

Parameters :

calibratedVal

the desired target value.

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

lightsensor.calibrateFromPoints()

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

```
int calibrateFromPoints( double rawValues, double refValues)
```

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

lightsensor.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

lightsensor.get_advertisedValue()

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

```
String get_advertisedValue()
```

Returns :

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

lightsensor.get_currentRawValue()

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

```
double get_currentRawValue()
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

lightsensor.get_currentValue()

Returns the current measured value.

```
double get_currentValue( )
```

Returns :

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

lightsensor.get_errorMessage()

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

```
String get_errorMessage( )
```

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

Returns :

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

lightsensor.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

lightsensor.get_lightsensorDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns Y_HARDWAREID_INVALID.

lightsensor.get_highestValue()

Returns the maximal value observed.

```
double get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed

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

lightsensor.get_logicalName()

Returns the logical name of the light sensor.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the light sensor

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

lightsensor.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed

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

lightsensor.get_module()

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

```
YModule get_module( )
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

lightsensor.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

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

Returns :

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

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

lightsensor.get_unit()

Returns the measuring unit for the measured value.

```
String get_unit( )
```

Returns :

a string corresponding to the measuring unit for the measured value

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

lightsensor.get_userData()

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

```
Object get_userData( )
```

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

Returns :

the object stored previously by the caller.

lightsensor.isOnline()

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

```
boolean isOnline( )
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

lightsensor.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

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

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

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

Returns :

nothing : the result is provided to the callback.

lightsensor.nextLightSensor()

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

```
YLightSensor nextLightSensor()
```

Returns :

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

lightsensor.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.

lightsensor.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.

lightsensor.set_logicalName()

Changes the logical name of the light sensor.

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

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

Parameters :

newval a string corresponding to the logical name of the light sensor

Returns :

YAPI_SUCCESS if the call succeeds.

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

lightsensor.set_lowestValue()

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.

lightsensor.set_userData()

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.14. 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:

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

Global functions

yFindModule(func)

Allows you to find a module from its serial number or from its logical name.

yFirstModule()

Starts the enumeration of modules currently accessible.

YModule methods

module→describe()

Returns a descriptive text that identifies the module.

module→functionCount()

Returns the number of functions (beside the "module" interface) available on the module.

module→functionId(functionIndex)

Retrieves the hardware identifier of the *n*th function on the module.

module→functionName(functionIndex)

Retrieves the logical name of the *n*th function on the module.

module→functionValue(functionIndex)

Retrieves the advertised value of the *n*th function on the module.

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

Returns the error message of the last error with this module object.

module→get_errorType()

Returns the numerical error code of the last error with this module object.

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_functionDescriptor()

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

module→get_hardwareId()

Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

Returns the luminosity of the module informative leds (from 0 to 100).

module→get_persistentSettings()

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

Changes the number of USB interfaces used by the module.

module→**set_userdata(data)**

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

module→**triggerFirmwareUpdate(secBeforeReboot)**

Schedules a module reboot into special firmware update mode.

YModule.FindModule()

Allows you to find a module from its serial number or from its logical name.

```
YModule FindModule( String func)
```

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

Parameters :

func a string containing either the serial number or the logical name of the desired module

Returns :

a `YModule` object allowing you to drive the module or get additional information on the module.

YModule.FirstModule()

Starts the enumeration of modules currently accessible.

```
YModule FirstModule()
```

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

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

module.describe()

Returns a descriptive text that identifies the module.

```
String describe()
```

The text may include either the logical name or the serial number of the module.

Returns :

a string that describes the module

Returns the number of functions (beside the "module" interface) available on the module.

Returns :

the number of functions on the module

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

Retrieves the hardware identifier of the *n*th function on the module.

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

Retrieves the logical name of the *n*th function on the module.

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

Retrieves the advertised value of the *n*th function on the module.

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

module.get_beacon()

Returns the state of the localization beacon.

```
int get_beacon( )
```

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module.get_errorMessage()

Returns the error message of the last error with this module object.

```
String get_errorMessage( )
```

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

Returns :

a string corresponding to the last error message that occurred while using this module object

module.get_errorType()

Returns the numerical error code of the last error with this module object.

```
int get_errorType( )
```

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

Returns :

a number corresponding to the code of the last error that occurred while using this module object

module.get_firmwareRelease()

Returns the version of the firmware embedded in the module.

```
String get_firmwareRelease( )
```

Returns :

a string corresponding to the version of the firmware embedded in the module

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

module.get_moduleDescriptor()

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 unique hardware identifier of the module.

The unique hardware identifier is made of the device serial number followed by string `".module"`.

Returns :

a string that uniquely identifies the module

Returns the icon of the module.

The icon is a png image and does not exceed 1024 bytes.

Returns :

a binary buffer with module icon, in png format.

module.get_logicalName()

Returns the logical name of the module.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the module

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

module.get_luminosity()

Returns the luminosity of the module informative leds (from 0 to 100).

```
int get_luminosity( )
```

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.

module.get_persistentSettings()

Returns the current state of persistent module settings.

```
int get_persistentSettings( )
```

Returns :

a value among Y_PERSISTENTSETTINGS_LOADED, Y_PERSISTENTSETTINGS_SAVED and Y_PERSISTENTSETTINGS_MODIFIED corresponding to the current state of persistent module settings

On failure, throws an exception or returns Y_PERSISTENTSETTINGS_INVALID.

module.get_productId()

Returns the USB device identifier of the module.

```
int get_productId( )
```

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y_PRODUCTID_INVALID.

module.get_productName()

Returns the commercial name of the module, as set by the factory.

```
String get_productName( )
```

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y_PRODUCTNAME_INVALID.

module.get_productRelease()

Returns the hardware release version of the module.

```
int get_productRelease( )
```

Returns :

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.

module.get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

```
int get_rebootCountdown( )
```

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

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

`module.get_serialNumber()`

Returns the serial number of the module, as set by the factory.

```
String get_serialNumber()
```

Returns :

a string corresponding to the serial number of the module, as set by the factory

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

`module.get_upTime()`

Returns the number of milliseconds spent since the module was powered on.

```
long get_upTime()
```

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

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

`module.get_usbBandwidth()`

Returns the number of USB interfaces used by the module.

```
int get_usbBandwidth()
```

Returns :

either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

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

`module.get_usbCurrent()`

Returns the current consumed by the module on the USB bus, in milli-amps.

```
int get_usbCurrent()
```

Returns :

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

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

`module.get_userData()`

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

```
Object get_userData()
```

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

Returns :

the object stored previously by the caller.

module.isOnline()

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

```
boolean isOnline()
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

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

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

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

module.load()

Preloads the module cache with a specified validity duration.

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

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

Parameters :

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

Returns :

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

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity an integer corresponding to the validity of the loaded module parameters, in milliseconds
callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or `YAPI_SUCCESS`)
context caller-specific object that is passed as-is to the callback function

Returns :
nothing : the result is provided to the callback.

module.nextModule()

Continues the module enumeration started using `yFirstModule()`.

```
YModule nextModule()
```

Returns :
a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module.reboot()

Schedules a simple module reboot after the given number of seconds.

```
int reboot(int secBeforeReboot)
```

Parameters :
secBeforeReboot number of seconds before rebooting

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

module.revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

```
int revertFromFlash()
```

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

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` if the call succeeds.

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

module.set_beacon()

Turns on or off the module localization beacon.

```
int set_beacon( int newval)
```

Parameters :

newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :

YAPI_SUCCESS if the call succeeds.

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

module.set_logicalName()

Changes the logical name of the module.

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

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

Parameters :

newval a string corresponding to the logical name of the module

Returns :

YAPI_SUCCESS if the call succeeds.

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

module.set_luminosity()

Changes the luminosity of the module informative leds.

```
int set_luminosity( int newval)
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

YAPI_SUCCESS if the call succeeds.

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

module.set_usbBandwidth()

Changes the number of USB interfaces used by the module.

```
int set_usbBandwidth( int newval)
```

Parameters :

newval either Y_USBBANDWIDTH_SIMPLE or Y_USBBANDWIDTH_DOUBLE, according to the number of USB interfaces used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

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

module.set_userData()

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

```
void set_userdata( Object data)
```

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

Parameters :

data any kind of object to be stored

module.triggerFirmwareUpdate()

Schedules a module reboot into special firmware update mode.

```
int triggerFirmwareUpdate( int secBeforeReboot)
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS if the call succeeds.

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

3.15. 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:

```
import com.yoctopuce.YoctoAPI.YPressure;
```

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 descriptive text that identifies the function.

pressure→get_advertisedValue()

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

pressure→get_currentRawValue()

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

pressure→get_currentValue()

Returns the current measured value.

pressure→get_errorMessage()

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

pressure→get_errorType()

Returns the numerical error code of the latest error with this function.

pressure→get_functionDescriptor()

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

pressure→get_hardwareId()	Returns the unique hardware identifier of the function.
pressure→get_highestValue()	Returns the maximal value observed.
pressure→get_logicalName()	Returns the logical name of the pressure sensor.
pressure→get_lowestValue()	Returns the minimal value observed.
pressure→get_module()	Get the YModule object for the device on which the function is located.
pressure→get_module_async(callback, context)	Get the YModule object for the device on which the function is located (asynchronous version).
pressure→get_resolution()	Returns the resolution of the measured values.
pressure→get_unit()	Returns the measuring unit for the measured value.
pressure→get_userData()	Returns the value of the userData attribute, as previously stored using method set_userData.
pressure→isOnline()	Checks if the function is currently reachable, without raising any error.
pressure→isOnline_async(callback, context)	Checks if the function is currently reachable, without raising any error (asynchronous version).
pressure→load(msValidity)	Preloads the function cache with a specified validity duration.
pressure→load_async(msValidity, callback, context)	Preloads the function cache with a specified validity duration (asynchronous version).
pressure→nextPressure()	Continues the enumeration of pressure sensors started using yFirstPressure().
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_logicalName(newval)	Changes the logical name of the pressure sensor.
pressure→set_lowestValue(newval)	Changes the recorded minimal value observed.
pressure→set_userData(data)	Stores a user context provided as argument in the userData attribute of the function.

YPressure.FindPressure()

Retrieves a pressure sensor for a given identifier.

```
YPressure FindPressure( String func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the pressure sensor

Returns :

a `YPressure` object allowing you to drive the pressure sensor.

`YPressure.FirstPressure()`

Starts the enumeration of pressure sensors currently accessible.

```
YPressure FirstPressure ( )
```

Use the method `YPressure.nextPressure()` to iterate on next pressure sensors.

Returns :

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

`pressure.calibrateFromPoints()`

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

```
int calibrateFromPoints( double rawValues, 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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

`pressure.describe()`

Returns a descriptive text that identifies the function.

```
String describe ( )
```


The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

pressure.get_advertisedValue()

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

```
String get_advertisedValue()
```

Returns :

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

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

pressure.get_currentRawValue()

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

```
double get_currentRawValue()
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

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

pressure.get_currentValue()

Returns the current measured value.

```
double get_currentValue()
```

Returns :

a floating point number corresponding to the current measured value

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

pressure.get_errorMessage()

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

```
String get_errorMessage()
```

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

Returns :

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

pressure.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

pressure.get_pressureDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

pressure.get_highestValue()

Returns the maximal value observed.

```
double get_highestValue()
```

Returns :

a floating point number corresponding to the maximal value observed

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

pressure.get_logicalName()

Returns the logical name of the pressure sensor.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the pressure sensor

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

pressure.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue()
```

Returns :

a floating point number corresponding to the minimal value observed

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

pressure.get_module()

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

`YModule get_module()`

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

Returns :
an instance of `YModule`

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

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

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

Returns :
nothing : the result is provided to the callback.

`pressure.get_resolution()`

Returns the resolution of the measured values.

`double get_resolution()`

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

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

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

`pressure.get_unit()`

Returns the measuring unit for the measured value.

`String get_unit()`

Returns :
a string corresponding to the measuring unit for the measured value

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

`pressure.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.

pressure.isOnline()

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

```
boolean isOnline()
```

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

pressure.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pressure.nextPressure()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

```
YPressure nextPressure()
```

Returns :

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

pressure.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.
-

pressure.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.

pressure.set_logicalName()

Changes the logical name of the pressure sensor.

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

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

Parameters :

newval a string corresponding to the logical name of the pressure sensor

Returns :

YAPI_SUCCESS if the call succeeds.

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

pressure.set_lowestValue()

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.

pressure.set_userData()

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

```
void set_userData( Object data)
```

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

Parameters :

data any kind of object to be stored

3.16. 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:

```
import com.yoctopuce.YoctoAPI.YRelay;
```

Global functions
yFindRelay(func) Retrieves a relay for a given identifier.
yFirstRelay() Starts the enumeration of relays currently accessible.
YRelay methods
relay→describe() Returns a descriptive text that identifies the function.
relay→get_advertisedValue() Returns the current value of the relay (no more than 6 characters).
relay→get_errorMessage() Returns the error message of the latest error with this function.

relay→get_errorType()	Returns the numerical error code of the latest error with this function.
relay→get_functionDescriptor()	Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
relay→get_hardwareId()	Returns the unique hardware identifier of the function.
relay→get_logicalName()	Returns the logical name of the relay.
relay→get_module()	Get the <code>YModule</code> object for the device on which the function is located.
relay→get_module_async(callback, context)	Get the <code>YModule</code> object for the device on which the function is located (asynchronous version).
relay→get_output()	Returns the output state of the relay, when used as a simple switch (single throw).
relay→get_pulseTimer()	Returns the number of milliseconds remaining before the relay is returned to idle position (state A), during a measured pulse generation.
relay→get_state()	Returns the state of the relay (A for the idle position, B for the active position).
relay→get_userData()	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
relay→isOnline()	Checks if the function is currently reachable, without raising any error.
relay→isOnline_async(callback, context)	Checks if the function is currently reachable, without raising any error (asynchronous version).
relay→load(msValidity)	Preloads the function cache with a specified validity duration.
relay→load_async(msValidity, callback, context)	Preloads the function cache with a specified validity duration (asynchronous version).
relay→nextRelay()	Continues the enumeration of relays started using <code>yFirstRelay()</code> .
relay→pulse(ms_duration)	Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).
relay→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
relay→set_logicalName(newval)	Changes the logical name of the relay.
relay→set_output(newval)	Changes the output state of the relay, when used as a simple switch (single throw).
relay→set_state(newval)	Changes the state of the relay (A for the idle position, B for the active position).
relay→set_userData(data)	Stores a user context provided as argument in the <code>userData</code> attribute of the function.

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.describe()

Returns a descriptive text that identifies the function.

```
String describe( )
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

relay.get_advertisedValue()

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

```
String get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the relay (no more than 6 characters)

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

relay.get_errorMessage()

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

```
String get_errorMessage()
```

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

Returns :

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

relay.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

relay.get_relayDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function. On failure, throws an exception or returns Y_HARDWAREID_INVALID.

relay.get_logicalName()

Returns the logical name of the relay.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the relay

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

relay.get_module()

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

```
YModule get_module( )
```

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

Returns :
an instance of `YModule`

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

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

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

Returns :
nothing : the result is provided to the callback.

relay.get_output()

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

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

relay.get_pulseTimer()

Returns the number of milliseconds remaining before the relay 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 relay is returned to idle position (state A), during a measured pulse generation

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

relay.get_state()

Returns the state of the relay (A for the idle position, B for the active position).

```
int get_state( )
```

Returns :
either `Y_STATE_A` or `Y_STATE_B`, according to the state of the relay (A for the idle position, B for the active position)

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

relay.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.

relay.isOnline()

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

boolean `isOnline()`

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

Returns :

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

relay.load()

Preloads the function 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.

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

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

relay.nextRelay()

Continues the enumeration of relays started using `yFirstRelay()`.

```
YRelay nextRelay( )
```

Returns :

a pointer to a `YRelay` object, corresponding to a relay currently online, or a `null` pointer if there are no more relays to enumerate.

relay.pulse()

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
int pulse( int ms_duration)
```

Parameters :

- ms_duration** pulse duration, in milliseconds

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

relay.registerValueCallback()

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.

relay.set_logicalName()

Changes the logical name of the relay.

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

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

Parameters :

newval a string corresponding to the logical name of the relay

Returns :

YAPI_SUCCESS if the call succeeds.

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

relay.set_output()

Changes the output state of the relay, 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 relay, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

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

relay.set_state()

Changes the state of the relay (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 relay (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

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

relay.set_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

3.17. 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:

```
import com.yoctopuce.YoctoAPI.YServo;
```

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 descriptive text that identifies the function.

servo→get_advertisedValue()

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

servo→get_errorMessage()

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

servo→get_errorType()

Returns the numerical error code of the latest error with this function.

servo→get_functionDescriptor()

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

servo→get_hardwareId()

Returns the unique hardware identifier of the function.

servo→get_logicalName()

Returns the logical name of the servo.

servo→get_module()

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

servo→get_module_async(callback, context)

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

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

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

servo→isOnline()

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

servo→isOnline_async(callback, context)

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

servo→load(msValidity)

Preloads the function cache with a specified validity duration.

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

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

servo→**move**(**target**, **ms_duration**)

Performs a smooth move at constant speed toward a given position.

servo→**nextServo**()

Continues the enumeration of servos started using `yFirstServo()`.

servo→**registerValueCallback**(**callback**)

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

servo→**set_logicalName**(**newval**)

Changes the logical name of the servo.

servo→**set_neutral**(**newval**)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→**set_position**(**newval**)

Changes immediately the servo driving position.

servo→**set_range**(**newval**)

Changes the range of use of the servo, specified in per cents.

servo→**set_userData**(**data**)

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

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()

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 descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

`servo.get_advertisedValue()`

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

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the servo (no more than 6 characters)

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

`servo.get_errorMessage()`

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

```
String get_errorMessage()
```

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

Returns :

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

`servo.get_errorType()`

Returns the numerical error code of the latest error with this function.

```
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 function object

`servo.get_servoDescriptor()`

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

`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()`

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

```
YModule get_module( )
```

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

Returns :

an instance of `YModule`

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

`servo.get_neutral()`

Returns the duration in microseconds of a neutral pulse for the servo.

```
int get_neutral( )
```

Returns :

an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns `Y_NEUTRAL_INVALID`.

servo.get_position()

Returns the current servo position.

```
int get_position()
```

Returns :

an integer corresponding to the current servo position

On failure, throws an exception or returns `Y_POSITION_INVALID`.

servo.get_range()

Returns the current range of use of the servo.

```
int get_range()
```

Returns :

an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns `Y_RANGE_INVALID`.

servo.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()

Checks if the function is currently reachable, without raising any error.

```
boolean isOnline()
```

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :

`true` if the function can be reached, and `false` otherwise

Checks if the function is currently reachable, without raising any error (asynchronous version).

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

servo.load()

Preloads the function 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.

Preloads the function cache with a specified validity duration (asynchronous version).

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

servo.move()

Performs a smooth move at constant speed toward a given position.

```
int move( int target, int ms_duration)
```

Parameters :

target new position at the end of the move

ms_duration total duration of the move, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo.nextServo()

Continues the enumeration of servos started using `yFirstServo()`.

`YServo nextServo()`

Returns :

a pointer to a `YServo` object, corresponding to a servo currently online, or a `null` pointer if there are no more servos to enumerate.

servo.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

`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.

servo.set_logicalName()

Changes the logical name of the servo.

`int set_logicalName(String newval)`

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the servo

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo.set_neutral()

Changes the duration of the pulse corresponding to the neutral position of the servo.

`int set_neutral(int newval)`

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

`servo.set_position()`

Changes immediately the servo driving position.

```
int set_position( int newval)
```

Parameters :

newval an integer corresponding to immediately the servo driving position

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

`servo.set_range()`

Changes the range of use of the servo, specified in per cents.

```
int set_range( int newval)
```

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms], When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the range of use of the servo, specified in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

`servo.set_userData()`

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

3.18. 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:

```
import com.yoctopuce.YoctoAPI.YTemperature;
```

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 descriptive text that identifies the function.

temperature→get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

temperature→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

temperature→get_currentValue()

Returns the current measured value.

temperature→get_errorMessage()

Returns the error message of the latest error with this function.

temperature→get_errorType()

Returns the numerical error code of the latest error with this function.

temperature→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

temperature→get_hardwareId()

Returns the unique hardware identifier of the function.

temperature→get_highestValue()

Returns the maximal value observed.

temperature→get_logicalName()

Returns the logical name of the temperature sensor.

temperature→get_lowestValue()

Returns the minimal value observed.

temperature→get_module()

Get the `YModule` object for the device on which the function is located.

temperature→get_module_async(callback, context)

Get the `YModule` object for the device on which the function is located (asynchronous version).

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 measured value.

temperature→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

temperature→isOnline()

Checks if the function is currently reachable, without raising any error.

temperature→isOnline_async(callback, context)

Checks if the function is currently reachable, without raising any error (asynchronous version).

temperature→load(msValidity)

Preloads the function cache with a specified validity duration.

temperature→load_async(msValidity, callback, context)

Preloads the function cache with a specified validity duration (asynchronous version).

temperature→nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

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_logicalName(newval)

Changes the logical name of the temperature sensor.

temperature→set_lowestValue(newval)

Changes the recorded minimal value observed.

temperature→set_sensorType(newval)

Modify the temperature sensor type.

temperature→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

YTemperature.FindTemperature()

Retrieves a temperature sensor for a given identifier.

```
YTemperature FindTemperature( String func)
```

The identifier can be specified using several formats:

- `FunctionLogicalName`
- `ModuleSerialNumber.FunctionIdentifier`
- `ModuleSerialNumber.FunctionLogicalName`
- `ModuleLogicalName.FunctionIdentifier`
- `ModuleLogicalName.FunctionLogicalName`

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the temperature sensor

Returns :

a `YTemperature` object allowing you to drive the temperature sensor.

YTemperature.FirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

```
YTemperature FirstTemperature()
```

Use the method `YTemperature.nextTemperature()` to iterate on next temperature sensors.

Returns :

a pointer to a `YTemperature` object, corresponding to the first temperature sensor currently online, or a `null` pointer if there are none.

temperature.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( double rawValues, 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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature.describe()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

temperature.get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the temperature sensor (no more than 6 characters)

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

temperature.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
double get_currentRawValue()
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

temperature.get_currentValue()

Returns the current measured value.

```
double get_currentValue()
```

Returns :

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

temperature.get_errorMessage()

Returns the error message of the latest error with this function.

```
String get_errorMessage()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this function object

temperature.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

temperature.get_temperatureDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns Y_HARDWAREID_INVALID.

temperature.get_highestValue()

Returns the maximal value observed.

```
double get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

temperature.get_logicalName()

Returns the logical name of the temperature sensor.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the temperature sensor

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

temperature.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

temperature.get_module()

Get the `YModule` object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

Get the `YModule` object for the device on which the function is located (asynchronous version).

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

temperature.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

The resolution corresponds to the numerical precision of the values, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

temperature.get_sensorType()

Returns the temperature sensor type.

```
int get_sensorType( )
```

Returns :

a value among `Y_SENSORTYPE_DIGITAL`, `Y_SENSORTYPE_TYPE_K`, `Y_SENSORTYPE_TYPE_E`, `Y_SENSORTYPE_TYPE_J`, `Y_SENSORTYPE_TYPE_N`, `Y_SENSORTYPE_TYPE_R`, `Y_SENSORTYPE_TYPE_S` and `Y_SENSORTYPE_TYPE_T` corresponding to the temperature sensor type

On failure, throws an exception or returns `Y_SENSORTYPE_INVALID`.

temperature.get_unit()

Returns the measuring unit for the measured value.

```
String get_unit( )
```

Returns :

a string corresponding to the measuring unit for the measured value

On failure, throws an exception or returns `Y_UNIT_INVALID`.

temperature.get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

```
Object get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

temperature.isOnline()

Checks if the function is currently reachable, without raising any error.

```
boolean isOnline( )
```

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :

`true` if the function can be reached, and `false` otherwise

Checks if the function is currently reachable, without raising any error (asynchronous version).

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

temperature.load()

Preloads the function 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.

Preloads the function cache with a specified validity duration (asynchronous version).

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

temperature.nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

```
YTemperature nextTemperature( )
```

Returns :

a pointer to a YTemperature object, corresponding to a temperature sensor currently online, or a null pointer if there are no more temperature sensors to enumerate.

temperature.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.

temperature.set_highestValue()

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature.set_logicalName()

Changes the logical name of the temperature sensor.

```
int set_logicalName( String newval)
```

You can use yCheckLogicalName() prior to this call to make sure that your parameter is valid. Remember to call the saveToFlash() method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the temperature sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature.set_lowestValue()

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature.set_sensorType()

Modify the temperature sensor type.

```
int set_sensorType( int newval)
```

This function is used to to define the type of thermo couple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among Y_SENSOR_TYPE_DIGITAL, Y_SENSOR_TYPE_K, Y_SENSOR_TYPE_E, Y_SENSOR_TYPE_J, Y_SENSOR_TYPE_N, Y_SENSOR_TYPE_R, Y_SENSOR_TYPE_S and Y_SENSOR_TYPE_T

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.

```
void set_userData( Object data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.19. 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:

```
import com.yoctopuce.YoctoAPI.YVoltage;
```

Global functions

yFindVoltage(func)

Retrieves a voltage sensor for a given identifier.

yFirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

YVoltage methods

voltage→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voltage→describe()

Returns a descriptive text that identifies the function.

voltage→get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).
voltage→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
voltage→get_currentValue() Returns the current measured value.
voltage→get_errorMessage() Returns the error message of the latest error with this function.
voltage→get_errorType() Returns the numerical error code of the latest error with this function.
voltage→get_functionDescriptor() Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
voltage→get_hardwareId() Returns the unique hardware identifier of the function.
voltage→get_highestValue() Returns the maximal value observed.
voltage→get_logicalName() Returns the logical name of the voltage sensor.
voltage→get_lowestValue() Returns the minimal value observed.
voltage→get_module() Get the <code>YModule</code> object for the device on which the function is located.
voltage→get_module_async(callback, context) Get the <code>YModule</code> object for the device on which the function is located (asynchronous version).
voltage→get_resolution() Returns the resolution of the measured values.
voltage→get_unit() Returns the measuring unit for the measured value.
voltage→get_userData() Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
voltage→isOnline() Checks if the function is currently reachable, without raising any error.
voltage→isOnline_async(callback, context) Checks if the function is currently reachable, without raising any error (asynchronous version).
voltage→load(msValidity) Preloads the function cache with a specified validity duration.
voltage→load_async(msValidity, callback, context) Preloads the function cache with a specified validity duration (asynchronous version).
voltage→nextVoltage() Continues the enumeration of voltage sensors started using <code>yFirstVoltage()</code> .
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_logicalName**(newval)

Changes the logical name of the voltage sensor.

voltage→**set_lowestValue**(newval)

Changes the recorded minimal value observed.

voltage→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

YVoltage.FindVoltage()

Retrieves a voltage sensor for a given identifier.

```
YVoltage FindVoltage( String func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

```
YVoltage FirstVoltage( )
```

Use the method `YVoltage.nextVoltage()` to iterate on next voltage sensors.

Returns :

a pointer to a `YVoltage` object, corresponding to the first voltage sensor currently online, or a `null` pointer if there are none.

voltage.calibrateFromPoints()

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( double rawValues, 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 lineat 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()

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

voltage.get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the voltage sensor (no more than 6 characters)

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

voltage.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
double get_currentRawValue()
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

voltage.get_currentValue()

Returns the current measured value.

```
double get_currentValue()
```

Returns :

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

voltage.get_errorMessage()

Returns the error message of the latest error with this function.

```
String get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this function object

voltage.get_errorType()

Returns the numerical error code of the latest error with this function.

```
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 function object

voltage.get_voltageDescriptor()

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function. On failure, throws an exception or returns Y_HARDWAREID_INVALID.

voltage.get_highestValue()

Returns the maximal value observed.

```
double get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

voltage.get_logicalName()

Returns the logical name of the voltage sensor.

```
String get_logicalName( )
```

Returns :

a string corresponding to the logical name of the voltage sensor

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

voltage.get_lowestValue()

Returns the minimal value observed.

```
double get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voltage.get_module()

Get the `YModule` object for the device on which the function is located.

```
YModule get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

Get the `YModule` object for the device on which the function is located (asynchronous version).

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage.get_resolution()

Returns the resolution of the measured values.

```
double get_resolution( )
```

The resolution corresponds to the numerical precision of the values, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voltage.get_unit()

Returns the measuring unit for the measured value.

```
String get_unit( )
```

Returns :

a string corresponding to the measuring unit for the measured value

On failure, throws an exception or returns `Y_UNIT_INVALID`.

`voltage.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.

`voltage.isOnline()`

Checks if the function is currently reachable, without raising any error.

boolean `isOnline()`

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :

`true` if the function can be reached, and `false` otherwise

Checks if the function is currently reachable, without raising any error (asynchronous version).

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

`voltage.load()`

Preloads the function 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.

Preloads the function cache with a specified validity duration (asynchronous version).

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage.nextVoltage()

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

```
YVoltage nextVoltage()
```

Returns :

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a null pointer if there are no more voltage sensors to enumerate.

voltage.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.

voltage.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.

voltage.set_logicalName()

Changes the logical name of the voltage sensor.

```
int set_logicalName( String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage.set_lowestValue()

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_userData()

Stores a user context provided as argument in the `userData` attribute of the function.

```
void set_userData( Object data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.20. 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:

```
import com.yoctopuce.YoctoAPI.YVSource;
```

Global functions**yFindVSource(func)**

Retrieves a voltage source for a given identifier.

yFirstVSource()

Starts the enumeration of voltage sources currently accessible.

YVSource methods

vsource→describe()

Returns a descriptive text that identifies the function.

vsource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

vsource→get_errorMessage()

Returns the error message of the latest error with this function.

vsource→get_errorType()

Returns the numerical error code of the latest error with this function.

vsource→get_extPowerFailure()

Return true if external power supply voltage is too low.

vsource→get_failure()

Return true if the module is in failure mode.

vsource→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

vsource→get_hardwareId()

Returns the unique hardware identifier of the function.

vsource→get_logicalName()

Returns the logical name of the voltage source.

vsource→get_module()

Get the `YModule` object for the device on which the function is located.

vsource→get_module_async(callback, context)

Get the `YModule` object for the device on which the function is located (asynchronous version).

vsource→get_overCurrent()

Return true if the appliance connected to the device is too greedy .

vsource→get_overHeat()

Return TRUE if the module is overheating.

vsource→get_overLoad()

Return true if the device is not able to maintaint the requested voltage output .

vsource→get_regulationFailure()

Return true if the voltage output is too high regarding the requested voltage .

vsource→get_unit()

Returns the measuring unit for the voltage.

vsource→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

vsource→get_voltage()

Returns the voltage output command (mV)

vsource→isOnline()

Checks if the function is currently reachable, without raising any error.

vsource→isOnline_async(callback, context)

Checks if the function is currently reachable, without raising any error (asynchronous version).

vsource→load(msValidity)

	Preloads the function cache with a specified validity duration.
<code>vsource→load_async(msValidity, callback, context)</code>	Preloads the function cache with a specified validity duration (asynchronous version).
<code>vsource→nextVSource()</code>	Continues the enumeration of voltage sources started using <code>yFirstVSource()</code> .
<code>vsource→pulse(voltage, ms_duration)</code>	Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.
<code>vsource→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>vsource→reset()</code>	Resets the device Output.
<code>vsource→set_logicalName(newval)</code>	Changes the logical name of the voltage source.
<code>vsource→set_userData(data)</code>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<code>vsource→set_voltage(newval)</code>	Tunes the device output voltage (milliVolts).
<code>vsource→voltageMove(target, ms_duration)</code>	Performs a smooth move at constant speed toward a given value.

`YVSource.FindVSource()`

Retrieves a voltage source for a given identifier.

```
YVSource FindVSource( String func)
```

The identifier can be specified using several formats:

- `FunctionLogicalName`
- `ModuleSerialNumber.FunctionIdentifier`
- `ModuleSerialNumber.FunctionLogicalName`
- `ModuleLogicalName.FunctionIdentifier`
- `ModuleLogicalName.FunctionLogicalName`

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

`func` a string that uniquely characterizes the voltage source

Returns :

a `YVSource` object allowing you to drive the voltage source.

`YVSource.FirstVSource()`

Starts the enumeration of voltage sources currently accessible.

```
YVSource FirstVSource( )
```

Use the method `YVSource.nextVSource()` to iterate on next voltage sources.

Returns :

a pointer to a `YVSource` object, corresponding to the first voltage source currently online, or a `null` pointer if there are none.

`vsource.describe()`

Returns a descriptive text that identifies the function.

```
String describe()
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

Returns :

a string that describes the function

`vsource.get_advertisedValue()`

Returns the current value of the voltage source (no more than 6 characters).

```
String get_advertisedValue()
```

Returns :

a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

`vsource.get_errorMessage()`

Returns the error message of the latest error with this function.

```
String get_errorMessage()
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this function object

`vsource.get_errorType()`

Returns the numerical error code of the latest error with this function.

```
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 function object

`vsource.get_extPowerFailure()`

Return true if external power supply voltage is too low.

```
int get_extPowerFailure()
```

Returns :

either `Y_EXTPOWERFAILURE_FALSE` or `Y_EXTPOWERFAILURE_TRUE`

On failure, throws an exception or returns `Y_EXTPOWERFAILURE_INVALID`.

`vsource.get_failure()`

Return true if the module is in failure mode.

```
int get_failure()
```

More information can be obtained by testing `get_overheat`, `get_overcurrent` etc... When a error condition is met, the output voltage is set to zero and cannot be changed until the `reset()` function is called.

Returns :

either `Y_FAILURE_FALSE` or `Y_FAILURE_TRUE`

On failure, throws an exception or returns `Y_FAILURE_INVALID`.

`vsource.getSourceDescriptor()`

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 unique hardware identifier of the function.

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

Returns :

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

`vsource.get_logicalName()`

Returns the logical name of the voltage source.

```
String get_logicalName()
```

Returns :

a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

`vsource.get_module()`

Get the `YModule` object for the device on which the function is located.

```
YModule get_module()
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

Get the `YModule` object for the device on which the function is located (asynchronous version).

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

`vsources.get_overCurrent()`

Return true if the appliance connected to the device is too greedy .

```
int get_overCurrent( )
```

Returns :

either `Y_OVERCURRENT_FALSE` or `Y_OVERCURRENT_TRUE`

On failure, throws an exception or returns `Y_OVERCURRENT_INVALID`.

`vsources.get_overHeat()`

Return TRUE if the module is overheating.

```
int get_overHeat( )
```

Returns :

either `Y_OVERHEAT_FALSE` or `Y_OVERHEAT_TRUE`

On failure, throws an exception or returns `Y_OVERHEAT_INVALID`.

`vsources.get_overLoad()`

Return true if the device is not able to maintain the requested voltage output .

```
int get_overLoad( )
```

Returns :

either `Y_OVERLOAD_FALSE` or `Y_OVERLOAD_TRUE`

On failure, throws an exception or returns `Y_OVERLOAD_INVALID`.

`vsources.get_regulationFailure()`

Return true if the voltage output is too high regarding the requested voltage .

```
int get_regulationFailure( )
```

Returns :

either `Y_REGULATIONFAILURE_FALSE` or `Y_REGULATIONFAILURE_TRUE`

On failure, throws an exception or returns `Y_REGULATIONFAILURE_INVALID`.

`vsource.get_unit()`

Returns the measuring unit for the voltage.

```
String get_unit()
```

Returns :

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

`vsource.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.

`vsource.get_voltage()`

Returns the voltage output command (mV)

```
int get_voltage()
```

Returns :

an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns `Y_VOLTAGE_INVALID`.

`vsource.isOnline()`

Checks if the function is currently reachable, without raising any error.

```
boolean isOnline()
```

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :

`true` if the function can be reached, and `false` otherwise

Checks if the function is currently reachable, without raising any error (asynchronous version).

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

vsource.load()

Preloads the function 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.

Preloads the function cache with a specified validity duration (asynchronous version).

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. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

vsource.nextVSource()

Continues the enumeration of voltage sources started using yFirstVSource().

```
YVSource nextVSource( )
```

Returns :

a pointer to a YVSource object, corresponding to a voltage source currently online, or a null pointer if there are no more voltage sources to enumerate.

vsource.pulse()

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

```
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.

vsource.reset()

Resets the device Output.

```
int reset()
```

This function must be called after any error condition. After an error condition, voltage output will be set to none and cannot be changed until this function is called.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource.set_logicalName()

Changes the logical name of the voltage source.

```
int set_logicalName( String newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage source

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource.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

vsource.set_voltage()

Tunes the device output voltage (milliVolts).

```
int set_voltage( int newval)
```

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource.voltageMove()

Performs a smooth move at constant speed toward a given value.

```
int voltageMove( int target, int ms_duration)
```

Parameters :

target new output value at end of transition, in milliVolts.

ms_duration transition duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

4. Index

A	
API	8
C	
calibrate	93
calibrateFromPoints	25
CheckLogicalName	9
D	
describe	16
DisableExceptions	9
E	
EnableExceptions	9
EnableUSBHost	9
F	
FindAnButton	16
FindCarbonDioxide	25
FindColorLed	33
FindCurrent	41
FindDataLogger	49
FindDualPower	63
FindHubPort	70
FindHumidity	77
FindLed	85
FindLightSensor	93
FindModule	102
FindPressure	111
FindRelay	120
FindServo	127
FindTemperature	135
FindVoltage	144
FindVSource	152
FirstAnButton	16
FirstCarbonDioxide	25
FirstColorLed	33
FirstCurrent	41
FirstDataLogger	50
FirstDualPower	64
FirstHubPort	70
FirstHumidity	78
FirstLed	86
FirstLightSensor	93
FirstModule	102
FirstPressure	112
FirstRelay	120

FirstServo	127
FirstTemperature	135
FirstVoltage	144
FirstVSource	152
forgetAllDataStreams	50
FreeAPI	10
functionCount	102
functionId	102
functionName	103
functionValue	103
G	
get_advertisedValue	16
get_analogCalibration	17
get_autoStart	51
get_averageValue	57
get_baudRate	71
get_beacon	103
get_blinking	86
get_calibratedValue	17
get_calibrationMax	17
get_calibrationMin	17
get_columnCount	60
get_columnNames	60
get_currentRawValue	26
get_currentRunIndex	51
get_currentValue	26
get_data	60
get_dataRows	61
get_dataRun	51
get_dataSamplesInterval	61
get_dataStreams	51
get_duration	57
get_enabled	71
get_errorMessage	17
get_errorType	18
get_extPowerFailure	153
get_extVoltage	65
get_failure	153
get_firmwareRelease	104
get_functionDescriptor	18
get_hardwareId	18
get_highestValue	27
get_hslColor	35
get_icon2d	104
get_isPressed	18
get_lastTimePressed	18
get_lastTimeReleased	18
get_logicalName	19
get_lowestValue	28
get_luminosity	87
get_maxValue	58
get_measureNames	52
get_minValue	58
get_module	19
get_module_async	19
get_neutral	129
get_oldestRunIndex	53
get_output	122
get_overCurrent	155
get_overHeat	155
get_overLoad	155
get_persistentSettings	105
get_portState	73
get_position	130

get_power	88
get_powerControl	66
get_powerState	66
get_productId	105
get_productName	105
get_productRelease	105
get_pulseTimer	122
get_range	130
get_rawValue	19
get_rebootCountdown	105
get_recording	53
get_regulationFailure	155
get_resolution	28
get_rgbColor	35
get_rgbColorAtPowerOn	36
get_rowCount	61
get_runIndex	61
get_sensitivity	20
get_sensorType	139
get_serialNumber	106
get_startTime	61
get_startTimeUTC	58
get_state	122
get_timeUTC	53
get_unit	29
get_upTime	106
get_usbBandwidth	106
get_usbCurrent	106
get_userData	20
get_valueCount	59
get_valueInterval	59
get_voltage	156
GetAPIVersion	10
GetTickCount	10
H	
HandleEvents	10
hslMove	36
I	
InitAPI	11
isOnline	20
isOnline_async	20
L	
load	21
load_async	21
M	
move	131
N	
nextAnButton	21
nextCarbonDioxide	30
nextColorLed	37
nextCurrent	46
nextDataLogger	55
nextDualPower	68
nextHubPort	74
nextHumidity	82
nextLed	89
nextLightSensor	98
nextModule	108
nextPressure	117
nextRelay	124
nextServo	131
nextTemperature	140
nextVoltage	149
nextVSource	157

P	
pulse	124
R	
reboot	108
RegisterDeviceArrivalCallback	11
RegisterDeviceRemovalCallback	11
RegisterHub	11
RegisterLogFunction	12
registerValueCallback	21
reset	158
revertFromFlash	108
rgbMove	38
S	
saveToFlash	108
set_analogCalibration	22
set_autoStart	55
set_beacon	108
set_blinking	90
set_calibrationMax	22
set_calibrationMin	22
set_enabled	75
set_highestValue	30
set_hslColor	38
set_logicalName	22
set_lowestValue	31
set_luminosity	90
set_neutral	132
set_output	125
set_position	133
set_power	91
set_powerControl	68
set_range	133
set_recording	56
set_rgbColor	39
set_rgbColorAtPowerOn	39
set_sensitivity	23
set_sensorType	142
set_state	125
set_timeUTC	56
set_usbBandwidth	109
set_userData	23
set_valueInterval	59
set_voltage	159
SetDelegate	12
SetTimeout	12
Sleep	13
T	
triggerFirmwareUpdate	110
U	
UnregisterHub	13
UpdateDeviceList	13
UpdateDeviceList_async	13
V	
voltageMove	159
Y	
YAnButton	14
YCarbonDioxide	23
YColorLed	31
YCurrent	39
YDataLogger	48
YDataRun	57
YDataStream	59
YDualPower	62
YHubPort	69

YHumidity	76
YLed	84
YLightSensor	91
YModule	100
YPressure	110
YRelay	118
YServo	126
YTemperature	133
YVoltage	142
YVSource	150