



C++ API Reference

Table of contents

1. Introduction	1
2. Using Yocto-Demo with C++	3
2.1. Control of the Led function	3
2.2. Control of the module part	5
2.3. Error handling	7
2.4. Integration variants for the C++ Yoctopuce library	8
Blueprint	12
3. Reference	12
3.1. General functions	13
3.2. Accelerometer function interface	33
3.3. AnButton function interface	75
3.4. CarbonDioxide function interface	113
3.5. ColorLed function interface	152
3.6. Compass function interface	181
3.7. Current function interface	221
3.8. DataLogger function interface	260
3.9. Formatted data sequence	291
3.10. Recorded data sequence	293
3.11. Unformatted data sequence	305
3.12. Digital IO function interface	320
3.13. Display function interface	364
3.14. DisplayLayer object interface	411
3.15. External power supply control interface	443
3.16. Files function interface	468
3.17. GenericSensor function interface	496
3.18. Gyroscope function interface	542
3.19. Yocto-hub port interface	593
3.20. Humidity function interface	618
3.21. Led function interface	657
3.22. LightSensor function interface	684
3.23. Magnetometer function interface	724
3.24. Measured value	766
3.25. Module control interface	772

3.26. Network function interface	815
3.27. OS control	872
3.28. Power function interface	895
3.29. Pressure function interface	938
3.30. Pwm function interface	977
3.31. PwmPowerSource function interface	1015
3.32. Quaternion interface	1038
3.33. Real Time Clock function interface	1077
3.34. Reference frame configuration	1104
3.35. Relay function interface	1140
3.36. Sensor function interface	1176
3.37. Servo function interface	1215
3.38. Temperature function interface	1250
3.39. Tilt function interface	1291
3.40. Voc function interface	1330
3.41. Voltage function interface	1369
3.42. Voltage source function interface	1408
3.43. WakeUpMonitor function interface	1440
3.44. WakeUpSchedule function interface	1475
3.45. Watchdog function interface	1512
3.46. Wireless function interface	1557

Index	1587
--------------------	-------------

1. Introduction

This manual is intended to be used as a reference for Yoctopuce C++ 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 being used. For more informations regarding the purpose and the usage of a given device attribute, please refer to the extended discussion provided in the device-specific user manual.

2. Using Yocto-Demo with C++

C++ is not the simplest language to master. However, if you take care to limit yourself to its essential functionalities, this language can very well be used for short programs quickly coded, and it has the advantage of being easily ported from one operating system to another. Under Windows, all the examples and the project models are tested with Microsoft Visual Studio 2010 Express, freely available on the Microsoft web site¹. Under Mac OS X, all the examples and project models are tested with XCode 4, available on the App Store. Moreover, under Mac OS X and under Linux, you can compile the examples using a command line with GCC using the provided `GNUmakefile`. In the same manner under Windows, a `Makefile` allows you to compile examples using a command line, fully knowing the compilation and linking arguments.

Yoctopuce C++ libraries² are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from C++. The library is naturally also available as binary files, so that you can link it directly if you prefer.

You will soon notice that the C++ API defines many functions which return objects. You do not need to deallocate these objects yourself, the API does it automatically at the end of the application.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface. You will find in the last section of this chapter all the information needed to create a wholly new project linked with the Yoctopuce libraries.

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 C++ code snippet to use the Led function.

```
#include "yocto_api.h"
#include "yocto_led.h"

[...]
String errmsg;
YLed *led;

// Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg);
led = yFindLed("YCTOPOC1-123456.led");
```

¹ <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express>

² www.yoctopuce.com/EN/libraries.php

```
// 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.

yocto_api.h et yocto_led.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_led.h` is necessary to manage modules containing a led, such as Yocto-Demo.

yRegisterHub

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

yFindLed

The `yFindLed` 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.

```
YLed *led = yFindLed("YCTOPOC1-123456.led");
YLed *led = yFindLed("YCTOPOC1-123456.MyFunction");
YLed *led = yFindLed("MyModule.led");
YLed *led = yFindLed("MyModule.MyFunction");
YLed *led = yFindLed("MyFunction");
```

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

isOnline

The `isOnline()` method of the object returned by `yFindLed` 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 `yFindLed` allows you to turn on and off the led. The argument is `Y_POWER_ON` or `Y_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 C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-Demo** of the Yoctopuce library. If you prefer to work with your favorite text editor, open the file `main.cpp`, and type `make` to build the example when you are done.

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

```
#include "yocto_api.h"
#include "yocto_led.h"
#include <iostream>
#include <stdlib.h>

using namespace std;
```



```

static void usage(void)
{
    cout << "usage: demo <serial_number> [ on | off ]" << endl;
    cout << "      demo <logical_name> [ on | off ]" << endl;
    cout << "      demo any [ on | off ]          (use any discovered device)" <<
endl;
    u64 now = yGetTickCount();    // dirty active wait loop
    while (yGetTickCount()-now<3000);
    exit(1);
}

int main(int argc, const char * argv[])
{
    string errmsg;
    string target;
    YLed *led;
    string on_off;

    if(argc < 3) {
        usage();
    }
    target = (string) argv[1];
    on_off = (string) argv[2];

    // Setup the API to use local USB devices
    if(yRegisterHub("usb", errmsg) != YAPI_SUCCESS) {
        cerr << "RegisterHub error: " << errmsg << endl;
        return 1;
    }

    if(target == "any"){
        led = yFirstLed();
    }else{
        led = yFindLed(target + ".led");
    }
    if (led && led->isOnline()) {
        led->set_power(on_off == "on" ? Y_POWER_ON : Y_POWER_OFF);
    } else {
        cout << "Module not connected (check identification and USB cable)" << endl;
    }

    return 0;
}

```

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.

```

#include <iostream>
#include <stdlib.h>

#include "yocto_api.h"

using namespace std;

static void usage(const char *exe)
{
    cout << "usage: " << exe << " <serial or logical name> [ON/OFF]" << endl;
    exit(1);
}

int main(int argc, const char * argv[])
{
    string errmsg;

    // Setup the API to use local USB devices
    if(yRegisterHub("usb", errmsg) != YAPI_SUCCESS) {
        cerr << "RegisterHub error: " << errmsg << endl;
        return 1;
    }
}

```

```

if(argc < 2)
    usage(argv[0]);

YModule *module = yFindModule(argv[1]); // use serial or logical name

if (module->isOnline()) {
    if (argc > 2) {
        if (string(argv[2]) == "ON")
            module->set_beacon(Y_BEACON_ON);
        else
            module->set_beacon(Y_BEACON_OFF);
    }
    cout << "serial:      " << module->get_serialNumber() << endl;
    cout << "logical name: " << module->get_logicalName() << endl;
    cout << "luminosity:  " << module->get_luminosity() << endl;
    cout << "beacon:      ";
    if (module->get_beacon() == Y_BEACON_ON)
        cout << "ON" << endl;
    else
        cout << "OFF" << endl;
    cout << "upTime:      " << module->get_upTime()/1000 << " sec" << endl;
    cout << "USB current: " << module->get_usbCurrent() << " mA" << endl;
    cout << "Logs:" << endl << module->get_lastLogs() << endl;
} else {
    cout << argv[1] << " not connected (check identification and USB cable)"
        << endl;
}
return 0;
}

```

Each property `xxx` of the module can be read thanks to a method of type `get_xxxx()`, and properties which are not read-only can be modified with the help of the `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 `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 `saveToFlash()` method. Inversely, it is possible to force the module to forget its current settings by using the `revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

```

#include <iostream>
#include <stdlib.h>

#include "yocto_api.h"

using namespace std;

static void usage(const char *exe)
{
    cerr << "usage: " << exe << " <serial> <newLogicalName>" << endl;
    exit(1);
}

int main(int argc, const char * argv[])
{
    string      errmsg;

    // Setup the API to use local USB devices
    if(yRegisterHub("usb", errmsg) != YAPI_SUCCESS) {
        cerr << "RegisterHub error: " << errmsg << endl;
        return 1;
    }

    if(argc < 2)
        usage(argv[0]);

    YModule *module = yFindModule(argv[1]); // use serial or logical name

    if (module->isOnline()) {
        if (argc >= 3){

```

```

        string newname = argv[2];
        if (!yCheckLogicalName(newname)){
            cerr << "Invalid name (" << newname << ")" << endl;
            usage(argv[0]);
        }
        module->set_logicalName(newname);
        module->saveToFlash();
    }
    cout << "Current name: " << module->get_logicalName() << endl;
} else {
    cout << argv[1] << " not connected (check identification and USB cable)"
        << endl;
}
return 0;
}

```

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 `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 `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.

```

#include <iostream>

#include "yocto_api.h"

using namespace std;

int main(int argc, const char * argv[])
{
    string      errmsg;

    // Setup the API to use local USB devices
    if(YAPI::RegisterHub("usb", errmsg) != YAPI_SUCCESS) {
        cerr << "RegisterHub error: " << errmsg << endl;
        return 1;
    }

    cout << "Device list: " << endl;

    YModule *module = YModule::FirstModule();
    while (module != NULL) {
        cout << module->get_serialNumber() << " ";
        cout << module->get_productName() << endl;
        module = module->nextModule();
    }
    return 0;
}

```

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. The only way to prevent this is to implement one of the two error handling techniques described below.

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

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

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

When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errorMessage()` methods. Their returned values contain the same information as in the exceptions when they are active.

2.4. Integration variants for the C++ Yoctopuce library

Depending on your needs and on your preferences, you can integrate the library into your projects in several distinct manners. This section explains how to implement the different options.

Integration in source format

Integrating all the sources of the library into your projects has several advantages:

- It guaranties the respect of the compilation conventions of your project (32/64 bits, inclusion of debugging symbols, unicode or ASCII characters, etc.);
- It facilitates debugging if you are looking for the cause of a problem linked to the Yoctopuce library;
- It reduces the dependencies on third party components, for example in the case where you would need to recompile this project for another architecture in many years;
- It does not require the installation of a dynamic library specific to Yoctopuce on the final system, everything is in the executable.

To integrate the source code, the easiest way is to simply include the `Sources` directory of your Yoctopuce library into your `IncludePath`, and to add all the files of this directory (including the sub-directory `yapi`) to your project.

For your project to build correctly, you need to link with your project the prerequisite system libraries, that is:

- For Windows: the libraries are added automatically
- For Mac OS X: **IOKit.framework** and **CoreFoundation.framework**
- For Linux: **libm**, **libpthread**, **libusb1.0**, and **libstdc++**

Integration as a static library

Integration of the Yoctopuce library as a static library is a simpler manner to build a small executable which uses Yoctopuce modules. You can quickly compile the program with a single command. You do not need to install a dynamic library specific to Yoctopuce, everything is in the executable.

To integrate the static Yoctopuce library to your project, you must include the `Sources` directory of the Yoctopuce library into your **IncludePath**, and add the sub-directory `Binaries/...` corresponding to your operating system into your **libPath**.

Then, for you project to build correctly, you need to link with your project the Yoctopuce library and the prerequisite system libraries:

- For Windows: **yocto-static.lib**
- For Mac OS X: **libyocto-static.a**, **IOKit.framework**, and **CoreFoundation.framework**
- For Linux: **libyocto-static.a**, **libm**, **libpthread**, **libusb1.0**, and **libstdc++**.

Note, under Linux, if you wish to compile in command line with GCC, it is generally advisable to link system libraries as dynamic libraries, rather than as static ones. To mix static and dynamic libraries on the same command line, you must pass the following arguments:

```
gcc (...) -Wl,-Bstatic -lyocto-static -Wl,-Bdynamic -lm -lpthread -lusb-1.0 -lstdc++
```

Integration as a dynamic library

Integration of the Yoctopuce library as a dynamic library allows you to produce an executable smaller than with the two previous methods, and to possibly update this library, if a patch reveals itself necessary, without needing to recompile the source code of the application. On the other hand, it is an integration mode which systematically requires you to copy the dynamic library on the target machine where the application will run (**yocto.dll** for Windows, **libyocto.so.1.0.1** for Mac OS X and Linux).

To integrate the dynamic Yoctopuce library to your project, you must include the `Sources` directory of the Yoctopuce library into your **IncludePath**, and add the sub-directory `Binaries/...` corresponding to your operating system into your **LibPath**.

Then, for you project to build correctly, you need to link with your project the dynamic Yoctopuce library and the prerequisite system libraries:

- For Windows: **yocto.lib**
- For Mac OS X: **libyocto**, **IOKit.framework**, and **CoreFoundation.framework**
- For Linux: **libyocto**, **libm**, **libpthread**, **libusb1.0**, and **libstdc++**.

With GCC, the command line to compile is simply:

```
gcc (...) -lyocto -lm -lpthread -lusb-1.0 -lstdc++
```


3. Reference

3.1. General functions

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

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

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

Global functions

yCheckLogicalName(name)

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

yDisableExceptions()

Disables the use of exceptions to report runtime errors.

yEnableExceptions()

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

yEnableUSBHost(osContext)

This function is used only on Android.

yFreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

yGetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

yGetTickCount()

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

yHandleEvents(errmsg)

Maintains the device-to-library communication channel.

yInitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

yPreregisterHub(url, errmsg)

Fault-tolerant alternative to RegisterHub().

yRegisterDeviceArrivalCallback(arrivalCallback)

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

yRegisterDeviceRemovalCallback(removalCallback)

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

yRegisterHub(url, errmsg)

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

yRegisterHubDiscoveryCallback(hubDiscoveryCallback)

3. Reference

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

yRegisterLogFunction(logfun)

Registers a log callback function.

ySelectArchitecture(arch)

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

ySetDelegate(object)

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

ySetTimeout(callback, ms_timeout, arguments)

Invoke the specified callback function after a given timeout.

ySleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

yTriggerHubDiscovery(errmsg)

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

yUnregisterHub(url)

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

yUpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName()**YAPI****yCheckLogicalName()**`yCheckLogicalName()`

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

```
bool yCheckLogicalName( const string& name)
```

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

Parameters :

name a string containing the name to check.

Returns :

`true` if the name is valid, `false` otherwise.

YAPI.DisableExceptions()

YAPI

yDisableExceptions()`yDisableExceptions()`

Disables the use of exceptions to report runtime errors.

```
void yDisableExceptions( )
```

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

YAPI.EnableExceptions()**YAPI****yEnableExceptions()****yEnableExceptions()**

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

```
void yEnableExceptions( )
```

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

YAPI.FreeAPI()

YAPI

yFreeAPI()yFreeAPI ()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

```
void yFreeAPI( )
```

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

YAPI.GetAPIVersion()**YAPI****yGetAPIVersion()**~~yGetAPIVersion()~~

Returns the version identifier for the Yoctopuce library in use.

string **yGetAPIVersion()**

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

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

Returns :

a character string describing the library version.

YAPI.GetTickCount()

YAPI

yGetTickCount()`yGetTickCount ()`

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

u64 **yGetTickCount()**

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

Returns :

a long integer corresponding to the millisecond counter.

YAPI.HandleEvents()**YAPI****yHandleEvents()**`yHandleEvents ()`

Maintains the device-to-library communication channel.

`YRETCODE yHandleEvents(string& errmsg)`

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

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

Parameters :

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

Returns :

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

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

Initializes the Yoctopuce programming library explicitly.

```
YRETCODE yInitAPI( int mode, string& errmsg)
```

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

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

Parameters :

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

Returns :

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

YAPI.PreregisterHub()**YAPI****yPreregisterHub()**`yPreregisterHub()`

Fault-tolerant alternative to `RegisterHub()`.

```
YRETCODE yPreregisterHub( const string& url, string& errmsg)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

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

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

```
void yRegisterDeviceArrivalCallback( yDeviceUpdateCallback arrivalCallback)
```

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

Parameters :

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

YAPI.RegisterDeviceRemovalCallback()
yRegisterDeviceRemovalCallback()
yRegisterDeviceRemovalCallback()

YAPI

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

```
void yRegisterDeviceRemovalCallback( yDeviceUpdateCallback removalCallback)
```

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

Parameters :

removalCallback a procedure taking a YModule parameter, or null

YAPI.RegisterHub()**YAPI****yRegisterHub()**`yRegisterHub()`

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

```
YRETCODE yRegisterHub( const string& url, string& errmsg)
```

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

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

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

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

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

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

```
http://username:password@adresse:port
```

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterHubDiscoveryCallback()
yRegisterHubDiscoveryCallback()
yRegisterHubDiscoveryCallback()

YAPI

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

```
void yRegisterHubDiscoveryCallback( YHubDiscoveryCallback hubDiscoveryCallback)
```

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

Parameters :

hubDiscoveryCallback a procedure taking two string parameter, or null

YAPI.RegisterLogFunction()

YAPI

yRegisterLogFunction()`yRegisterLogFunction()`

Registers a log callback function.

```
void yRegisterLogFunction( yLogFunction logfun)
```

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

Parameters :

logfun a procedure taking a string parameter, or null

YAPI.Sleep()**YAPI****ySleep()**`ySleep()`

Pauses the execution flow for a specified duration.

```
YRETCODE ySleep( unsigned ms_duration, string& errmsg)
```

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

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

Parameters :

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

Returns :

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

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

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

YRETCODE yTriggerHubDiscovery(string& errmsg)

Parameters :

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

Returns :

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

YAPI.UnregisterHub()**YAPI****yUnregisterHub()**`yUnregisterHub()`

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

```
void yUnregisterHub( const string& url)
```

Parameters :

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

YAPI.UpdateDeviceList()**YAPI****yUpdateDeviceList()**`yUpdateDeviceList()`

Triggers a (re)detection of connected Yoctopuce modules.

YRETCODE `yUpdateDeviceList(string& errmsg)`

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

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

Parameters :

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

Returns :

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

3.2. Accelerometer function interface

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

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

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

Global functions

yFindAccelerometer(func)

Retrieves an accelerometer for a given identifier.

yFirstAccelerometer()

Starts the enumeration of accelerometers currently accessible.

YAccelerometer methods

accelerometer→calibrateFromPoints(rawValues, refValues)

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

accelerometer→describe()

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

accelerometer→get_advertisedValue()

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

accelerometer→get_currentRawValue()

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

accelerometer→get_currentValue()

Returns the current value of the acceleration.

accelerometer→get_errorMessage()

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

accelerometer→get_errorType()

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

accelerometer→get_friendlyName()

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

accelerometer→get_functionDescriptor()

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

accelerometer→get_functionId()

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

accelerometer→get_hardwareId()

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

accelerometer→get_highestValue()

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

accelerometer→get_logFrequency()

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

accelerometer→get_logicalName()

Returns the logical name of the accelerometer.

accelerometer→get_lowestValue()

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

accelerometer→get_module()

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

accelerometer→get_module_async(callback, context)

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

accelerometer→get_recordedData(startTime, endTime)

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

accelerometer→get_reportFrequency()

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

accelerometer→get_resolution()

Returns the resolution of the measured values.

accelerometer→get_unit()

Returns the measuring unit for the acceleration.

accelerometer→get_userData()

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

accelerometer→get_xValue()

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

accelerometer→get_yValue()

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

accelerometer→get_zValue()

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

accelerometer→isOnline()

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

accelerometer→isOnline_async(callback, context)

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

accelerometer→load(msValidity)

Preloads the accelerometer cache with a specified validity duration.

accelerometer→loadCalibrationPoints(rawValues, refValues)

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

accelerometer→load_async(msValidity, callback, context)

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

accelerometer→nextAccelerometer()

Continues the enumeration of accelerometers started using yFirstAccelerometer().

accelerometer→registerTimedReportCallback(callback)

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

accelerometer→registerValueCallback(callback)

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

accelerometer→set_highestValue(newval)

Changes the recorded maximal value observed.

accelerometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

accelerometer→set_logicalName(newval)

Changes the logical name of the accelerometer.

accelerometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

accelerometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

accelerometer→set_resolution(newval)

Changes the resolution of the measured physical values.

accelerometer→set_userData(data)

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

accelerometer→wait_async(callback, context)

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

YAccelerometer.FindAccelerometer() yFindAccelerometer()yFindAccelerometer()

YAccelerometer

Retrieves an accelerometer for a given identifier.

```
YAccelerometer* yFindAccelerometer( const string& func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the accelerometer

Returns :

a `YAccelerometer` object allowing you to drive the accelerometer.

YAccelerometer.FirstAccelerometer()**YAccelerometer****yFirstAccelerometer()****yFirstAccelerometer()**

Starts the enumeration of accelerometers currently accessible.

YAccelerometer* **yFirstAccelerometer()**

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

Returns :

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

accelerometer→calibrateFromPoints()**YAccelerometer****accelerometer→calibrateFromPoints()**

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

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

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→describe()accelerometer→
describe()

YAccelerometer

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

string **describe()**

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

Returns :

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

accelerometer→get_advertisedValue()

YAccelerometer

accelerometer→advertisedValue()**accelerometer→**
get_advertisedValue()

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

string get_advertisedValue()

Returns :

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

accelerometer→**get_currentRawValue()****YAccelerometer****accelerometer**→**currentRawValue()****accelerometer**→**get_currentRawValue()**

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

double **get_currentRawValue()**

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

accelerometer→get_currentValue()

YAccelerometer

accelerometer→currentValue()**accelerometer→**

get_currentValue()

Returns the current value of the acceleration.

double get_currentValue()

Returns :

a floating point number corresponding to the current value of the acceleration

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

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

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

`string get_errorMessage()`

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

Returns :

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

accelerometer→get_errorType()

YAccelerometer

accelerometer→errorType()**accelerometer→**
get_errorType()

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

YRETCODE **get_errorType()**

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

Returns :

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

accelerometer→get_friendlyName()**YAccelerometer****accelerometer→friendlyName()****accelerometer→
get_friendlyName()**

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

`string` **get_friendlyName()**

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

Returns :

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

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

accelerometer→**get_functionDescriptor()**

YAccelerometer

accelerometer→**functionDescriptor()****accelerometer**

→**get_functionDescriptor()**

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

`YFUN_DESCR` **get_functionDescriptor()**

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

Returns :

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

accelerometer→**get_functionId()****YAccelerometer****accelerometer**→**functionId()****accelerometer**→
get_functionId()

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

string **get_functionId()**

For example `relay1`

Returns :

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

accelerometer→**get_hardwareId()**

YAccelerometer

accelerometer→**hardwareId()****accelerometer**→
get_hardwareId()

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

`string get_hardwareId()`

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

Returns :

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

accelerometer→get_highestValue()**YAccelerometer****accelerometer→highestValue()****accelerometer→**
get_highestValue()

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

double **get_highestValue()**

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

accelerometer→get_logFrequency()

YAccelerometer

accelerometer→logFrequency()accelerometer→

get_logFrequency()

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

string **get_logFrequency()**

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

accelerometer→**get_logicalName()****YAccelerometer****accelerometer**→**logicalName()****accelerometer**→
get_logicalName()

Returns the logical name of the accelerometer.

string **get_logicalName()**

Returns :

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

accelerometer→get_lowestValue()

YAccelerometer

accelerometer→lowestValue()**accelerometer→**
get_lowestValue()

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

double get_lowestValue()

Returns :

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

accelerometer→**get_module()****YAccelerometer****accelerometer**→**module()****accelerometer**→
get_module()

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

YModule * **get_module()**

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

Returns :

an instance of YModule

accelerometer→**get_recordedData()****YAccelerometer****accelerometer**→**recordedData()****accelerometer**→
get_recordedData()

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

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

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

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

Parameters :

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

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

Returns :

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

accelerometer→get_reportFrequency()**YAccelerometer****accelerometer→reportFrequency()****accelerometer→**
get_reportFrequency()

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

string **get_reportFrequency()**

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

accelerometer→get_resolution()

YAccelerometer

accelerometer→resolution()**accelerometer→**
get_resolution()

Returns the resolution of the measured values.

`double get_resolution()`

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

accelerometer→get_unit()**YAccelerometer****accelerometer→unit()****accelerometer→get_unit()**

Returns the measuring unit for the acceleration.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the acceleration

On failure, throws an exception or returns Y_UNIT_INVALID.

accelerometer→get_userdata()

YAccelerometer

accelerometer→userdata()**accelerometer→**
get_userdata()

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

```
void * get_userdata( )
```

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

Returns :

the object stored previously by the caller.

accelerometer→**get_xValue()****YAccelerometer****accelerometer**→**xValue()****accelerometer**→
get_xValue()

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

double **get_xValue()**

Returns :

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

On failure, throws an exception or returns Y_XVALUE_INVALID.

accelerometer→get_yValue()

YAccelerometer

accelerometer→yValue()**accelerometer→**
get_yValue()

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

double **get_yValue()**

Returns :

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

On failure, throws an exception or returns Y_YVALUE_INVALID.

accelerometer→get_zValue()**YAccelerometer****accelerometer→zValue()****accelerometer→**
get_zValue()

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

`double` **get_zValue()**

Returns :

a floating point number corresponding to the Z component of the acceleration, as a floating point number

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

accelerometer→**isOnline()****accelerometer**→
isOnline()

YAccelerometer

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

bool **isOnline()**

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

Returns :

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

accelerometer→load()**accelerometer→load()****YAccelerometer**

Preloads the accelerometer cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

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

Parameters :

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

Returns :

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

accelerometer→loadCalibrationPoints()**YAccelerometer****accelerometer→loadCalibrationPoints()**

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

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

```
accelerometer→nextAccelerometer()accelerometer  
→nextAccelerometer()
```

YAccelerometer

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

```
YAccelerometer * nextAccelerometer()
```

Returns :

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

accelerometer→registerTimedReportCallback()**YAccelerometer****accelerometer→****registerTimedReportCallback()**

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

```
int registerTimedReportCallback( YAccelerometerTimedReportCallback callback)
```

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

Parameters :

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

accelerometer→registerValueCallback()**YAccelerometer****accelerometer→registerValueCallback()**

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

```
int registerValueCallback( YAccelerometerValueCallback callback)
```

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

Parameters :

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

accelerometer→**set_highestValue()**

YAccelerometer

accelerometer→**setHighestValue()****accelerometer**→
set_highestValue()

Changes the recorded maximal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_logFrequency()**YAccelerometer****accelerometer→setLogFrequency()accelerometer
→set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_logicalName()

YAccelerometer

accelerometer→setLogicalName()**accelerometer→set_logicalName()**

Changes the logical name of the accelerometer.

```
int set_logicalName( const string& newval)
```

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

Parameters :

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

Returns :

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

accelerometer→set_lowestValue()**YAccelerometer****accelerometer→setLowestValue()****accelerometer→**
set_lowestValue()

Changes the recorded minimal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_reportFrequency()**YAccelerometer****accelerometer→setReportFrequency()****accelerometer→set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→**set_resolution()****YAccelerometer****accelerometer**→**setResolution()****accelerometer**→**set_resolution()**

Changes the resolution of the measured physical values.

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_userdata()

YAccelerometer

accelerometer→setUserData()**accelerometer→**
set_userdata()

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

```
void set_userdata( void* data)
```

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

Parameters :

data any kind of object to be stored

3.3. AnButton function interface

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

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

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

Global functions

yFindAnButton(func)

Retrieves an analog input for a given identifier.

yFirstAnButton()

Starts the enumeration of analog inputs currently accessible.

YAnButton methods

anbutton→describe()

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

anbutton→get_advertisedValue()

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

anbutton→get_analogCalibration()

Tells if a calibration process is currently ongoing.

anbutton→get_calibratedValue()

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

anbutton→get_calibrationMax()

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

anbutton→get_calibrationMin()

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

anbutton→get_errorMessage()

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

anbutton→get_errorType()

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

anbutton→get_friendlyName()

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

anbutton→get_functionDescriptor()

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

anbutton→get_functionId()

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

anbutton→get_hardwareId()

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

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

anbutton→get_module_async(callback, context)

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

anbutton→get_pulseCounter()

Returns the pulse counter value

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms)

anbutton→get_rawValue()

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

anbutton→get_sensitivity()

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

anbutton→get_userData()

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

anbutton→isOnline()

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

anbutton→isOnline_async(callback, context)

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

anbutton→load(msValidity)

Preloads the analog input cache with a specified validity duration.

anbutton→load_async(msValidity, callback, context)

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

anbutton→nextAnButton()

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

anbutton→registerValueCallback(callback)

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

anbutton→resetCounter()

Returns the pulse counter value as well as his timer

anbutton→set_analogCalibration(newval)

Starts or stops the calibration process.

anbutton→set_calibrationMax(newval)

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

anbutton→set_calibrationMin(newval)

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

anbutton→set_logicalName(newval)

Changes the logical name of the analog input.

anbutton→set_sensitivity(newval)

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

anbutton→set_userData(data)

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

anbutton→wait_async(callback, context)

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

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

Retrieves an analog input for a given identifier.

```
YAnButton* yFindAnButton( const string& func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the analog input

Returns :

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

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

YAnButton

Starts the enumeration of analog inputs currently accessible.

`YAnButton* yFirstAnButton()`

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

Returns :

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

anbutton→describe()**anbutton→describe()****YAnButton**

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

string **describe()**

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

Returns :

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

anbutton→**get_advertisedValue()****YAnButton****anbutton**→**advertisedValue()****anbutton**→
get_advertisedValue()

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

string **get_advertisedValue()**

Returns :

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

anbutton→**get_analogCalibration()**

YAnButton

anbutton→**analogCalibration()****anbutton**→
get_analogCalibration()

Tells if a calibration process is currently ongoing.

[Y_ANALOGCALIBRATION_enum](#) **get_analogCalibration()** ()

Returns :

either Y_ANALOGCALIBRATION_OFF or Y_ANALOGCALIBRATION_ON

On failure, throws an exception or returns Y_ANALOGCALIBRATION_INVALID.

anbutton→**get_calibratedValue()****YAnButton****anbutton**→**calibratedValue()****anbutton**→
get_calibratedValue()

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

int **get_calibratedValue()**

Returns :

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

On failure, throws an exception or returns Y_CALIBRATEDVALUE_INVALID.

anbutton→**get_calibrationMax()**

YAnButton

anbutton→**calibrationMax()****anbutton**→
get_calibrationMax()

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

int **get_calibrationMax()**

Returns :

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

On failure, throws an exception or returns Y_CALIBRATIONMAX_INVALID.

anbutton→get_calibrationMin()**YAnButton****anbutton→calibrationMin()****anbutton→**
get_calibrationMin()

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

int **get_calibrationMin()**

Returns :

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

On failure, throws an exception or returns Y_CALIBRATIONMIN_INVALID.

anbutton→**get_errorMessage()**

YAnButton

anbutton→**errorMessage()****anbutton**→
get_errorMessage()

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

string **get_errorMessage()**

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

Returns :

a string corresponding to the latest error message that occurred while using the analog input object

anbutton→**get_errorType()****YAnButton****anbutton**→**errorType()****anbutton**→
get_errorType()

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

YRETCODE **get_errorType()**

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

Returns :

a number corresponding to the code of the latest error that occurred while using the analog input object

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

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

string **get_friendlyName()**

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

Returns :

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

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

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

`YFUN_DESCR` **get_functionDescriptor()**

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

Returns :

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

anbutton→**get_functionId()**

YAnButton

anbutton→**functionId()****anbutton**→

get_functionId()

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

`string get_functionId()`

For example `relay1`

Returns :

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

anbutton→**get_hardwareId()**
anbutton→**hardwareId()****anbutton**→
get_hardwareId()

YAnButton

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

`string get_hardwareId()`

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

Returns :

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

anbutton→get_isPressed()

YAnButton

anbutton→isPressed()
anbutton→get_isPressed()

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

Y_ISPRESSED_enum get_isPressed()

Returns :

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

On failure, throws an exception or returns Y_ISPRESSED_INVALID.

anbutton→get_lastTimePressed()**YAnButton****anbutton→lastTimePressed()**
anbutton→get_lastTimePressed()

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

s64 **get_lastTimePressed()**

Returns :

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

On failure, throws an exception or returns Y_LASTTIMEPRESSED_INVALID.

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

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

s64 **get_lastTimeReleased()****Returns :**

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

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

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

Returns the logical name of the analog input.

string **get_logicalName()**

Returns :

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

anbutton→get_module()

YAnButton

anbutton→module()**anbutton→get_module()**

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

YModule * **get_module()**

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

Returns :

an instance of YModule

anbutton→**get_pulseCounter()**
anbutton→**pulseCounter()****anbutton**→
get_pulseCounter()

YAnButton

Returns the pulse counter value

s64 **get_pulseCounter()**

Returns :

an integer corresponding to the pulse counter value

On failure, throws an exception or returns Y_PULSECOUNTER_INVALID.

anbutton→get_pulseTimer()

YAnButton

anbutton→pulseTimer()
anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms)

s64 **get_pulseTimer()**

Returns :

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

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

anbutton→get_rawValue()**YAnButton****anbutton→rawValue()****anbutton→get_rawValue()**

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

int **get_rawValue()**

Returns :

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

On failure, throws an exception or returns Y_RAWVALUE_INVALID.

anbutton→get_sensitivity()

YAnButton

anbutton→sensitivity()
anbutton→get_sensitivity()

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

int get_sensitivity()

Returns :

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

On failure, throws an exception or returns Y_SENSITIVITY_INVALID.

anbutton→get_userdata()**YAnButton****anbutton→userData()****anbutton→get_userdata()**

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

```
void * get_userdata( )
```

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

Returns :

the object stored previously by the caller.

anbutton→isOnline()**anbutton→isOnline()**

YAnButton

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

`bool isOnline()`

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

Returns :

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

anbutton→load()**anbutton→load()****YAnButton**

Preloads the analog input cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

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

Parameters :

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

Returns :

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

anbutton→**nextAnButton()****anbutton**→
nextAnButton()

YAnButton

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()**anbutton→**
registerValueCallback()

YAnButton

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

```
int registerValueCallback( YAnButtonValueCallback callback)
```

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

Parameters :

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

anbutton→resetCounter()anbutton→
resetCounter ()

YAnButton

Returns the pulse counter value as well as his timer

int resetCounter()

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→set_analogCalibration()**YAnButton****anbutton→setAnalogCalibration()**
anbutton→set_analogCalibration()

Starts or stops the calibration process.

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

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

Parameters :

newval either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton→**set_calibrationMax()****YAnButton****anbutton**→**setCalibrationMax()****anbutton**→
set_calibrationMax()

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

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→**set_calibrationMin()****YAnButton****anbutton**→**setCalibrationMin()****anbutton**→
set_calibrationMin()

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

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→**set_logicalName()****YAnButton****anbutton**→**setLogicalName()****anbutton**→**set_logicalName()**

Changes the logical name of the analog input.

```
int set_logicalName( const string& newval)
```

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

Parameters :

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

Returns :

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

anbutton→**set_sensitivity()****YAnButton****anbutton**→**setSensitivity()****anbutton**→**set_sensitivity()**

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

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→set_userdata()

YAnButton

anbutton→setUserData()**anbutton→**

set_userdata()

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

```
void set_userdata( void* data)
```

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

Parameters :

data any kind of object to be stored

3.4. CarbonDioxide function interface

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

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

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

Global functions

yFindCarbonDioxide(func)

Retrieves a CO2 sensor for a given identifier.

yFirstCarbonDioxide()

Starts the enumeration of CO2 sensors currently accessible.

YCarbonDioxide methods

carbondioxide→calibrateFromPoints(rawValues, refValues)

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

carbondioxide→describe()

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

carbondioxide→get_advertisedValue()

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

carbondioxide→get_currentRawValue()

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

carbondioxide→get_currentValue()

Returns the current value of the CO2 concentration.

carbondioxide→get_errorMessage()

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

carbondioxide→get_errorType()

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

carbondioxide→get_friendlyName()

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

carbondioxide→get_functionDescriptor()

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

carbondioxide→get_functionId()

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

carbondioxide→get_hardwareId()

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

carbondioxide→get_highestValue()

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

carbondioxide→get_logFrequency()

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

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

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

carbondioxide→get_module()

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

carbondioxide→get_module_async(callback, context)

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

carbondioxide→get_recordedData(startTime, endTime)

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

carbondioxide→get_reportFrequency()

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_unit()

Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

carbondioxide→isOnline_async(callback, context)

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

carbondioxide→load(msValidity)

Preloads the CO2 sensor cache with a specified validity duration.

carbondioxide→loadCalibrationPoints(rawValues, refValues)

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

carbondioxide→load_async(msValidity, callback, context)

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

carbondioxide→nextCarbonDioxide()

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

carbondioxide→registerTimedReportCallback(callback)

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

carbondioxide→registerValueCallback(callback)

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

carbondioxide→set_highestValue(newval)

Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(newval)

Changes the logical name of the CO2 sensor.

carbondioxide→set_lowestValue(newval)

Changes the recorded minimal value observed.

carbondioxide→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

carbondioxide→set_resolution(newval)

Changes the resolution of the measured physical values.

carbondioxide→set_userData(data)

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

carbondioxide→wait_async(callback, context)

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

YCarbonDioxide.FindCarbonDioxide() yFindCarbonDioxide()

YCarbonDioxide

Retrieves a CO2 sensor for a given identifier.

```
YCarbonDioxide* yFindCarbonDioxide( const 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()
yFirstCarbonDioxide()**

YCarbonDioxide

Starts the enumeration of CO2 sensors currently accessible.

YCarbonDioxide* yFirstCarbonDioxide()

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

Returns :

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

carbondioxide→calibrateFromPoints()**YCarbonDioxide****carbondioxide→calibrateFromPoints()**

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

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<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()carbondioxide→
describe()

YCarbonDioxide

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

string **describe()**

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

Returns :

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

carbondioxide→get_advertisedValue()

YCarbonDioxide

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

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

string get_advertisedValue()

Returns :

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

carbondioxide→**get_currentRawValue()****YCarbonDioxide****carbondioxide**→**currentRawValue()****carbondioxide**→**get_currentRawValue()**

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

double **get_currentRawValue()**

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

carbondioxide→get_currentValue()

YCarbonDioxide

carbondioxide→currentValue()carbondioxide→

get_currentValue()

Returns the current value of the CO2 concentration.

double **get_currentValue**()

Returns :

a floating point number corresponding to the current value of the CO2 concentration

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

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

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

`string` **get_errorMessage()**

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

Returns :

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

carbondioxide→get_errorType()

YCarbonDioxide

carbondioxide→errorType()**carbondioxide→**
get_errorType()

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

YRETCODE **get_errorType()**

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

Returns :

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

carbondioxide→**get_friendlyName()****YCarbonDioxide****carbondioxide**→**friendlyName()****carbondioxide**→
get_friendlyName()

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

```
string get_friendlyName()
```

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

Returns :

a string that uniquely identifies the CO2 sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

carbondioxide→get_functionDescriptor()

YCarbonDioxide

carbondioxide→functionDescriptor()carbondioxide

→get_functionDescriptor()

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

YFUN_DESCR **get_functionDescriptor()**

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

Returns :

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

carbondioxide→**get_functionId()****YCarbonDioxide****carbondioxide**→**functionId()****carbondioxide**→
get_functionId()

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

`string` **get_functionId()**

For example `relay1`

Returns :

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

carbondioxide→get_hardwareId()

YCarbonDioxide

carbondioxide→hardwareId()**carbondioxide→**
get_hardwareId()

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

`string get_hardwareId()`

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

Returns :

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

carbondioxide→get_highestValue()**YCarbonDioxide****carbondioxide→highestValue()****carbondioxide→**
get_highestValue()

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

double **get_highestValue()**

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

carbondioxide→get_logFrequency()

YCarbonDioxide

carbondioxide→logFrequency()carbondioxide→
get_logFrequency()

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

string **get_logFrequency()**

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

carbondioxide→**get_logicalName()****YCarbonDioxide****carbondioxide**→**logicalName()****carbondioxide**→
get_logicalName()

Returns the logical name of the CO2 sensor.

string **get_logicalName()**

Returns :

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

carbondioxide→get_lowestValue()

YCarbonDioxide

carbondioxide→lowestValue()
carbondioxide→
get_lowestValue()

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

double **get_lowestValue()**

Returns :

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

carbondioxide→get_module()**YCarbonDioxide****carbondioxide→module()****carbondioxide→**
get_module()

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

YModule * **get_module()**

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

Returns :

an instance of YModule

carbondioxide→get_recordedData()**YCarbonDioxide****carbondioxide→recordedData()****carbondioxide→get_recordedData()**

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

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

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

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

Parameters :

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

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

Returns :

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

carbondioxide→get_reportFrequency()**YCarbonDioxide****carbondioxide→reportFrequency()****carbondioxide→**
get_reportFrequency()

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

string **get_reportFrequency()**

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

carbondioxide→get_resolution()

YCarbonDioxide

carbondioxide→resolution()**carbondioxide→**
get_resolution()

Returns the resolution of the measured values.

double get_resolution()

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

carbondioxide→get_unit()**YCarbonDioxide****carbondioxide→unit()****carbondioxide→get_unit()**

Returns the measuring unit for the CO2 concentration.

string **get_unit()**

Returns :

a string corresponding to the measuring unit for the CO2 concentration

On failure, throws an exception or returns Y_UNIT_INVALID.

carbondioxide→get_userdata()

YCarbonDioxide

carbondioxide→userdata()**carbondioxide→**
get_userdata()

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

```
void * 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()****carbondioxide**→
isOnline()

YCarbonDioxide

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

bool isOnline()

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

Returns :

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

carbondioxide→load()**carbondioxide→load()****YCarbonDioxide**

Preloads the CO2 sensor cache with a specified validity duration.

YRETCODE **load(int msValidity)**

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

Parameters :

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

Returns :

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

carbondioxide→loadCalibrationPoints()**YCarbonDioxide****carbondioxide→loadCalibrationPoints()**

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

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→nextCarbonDioxide()

YCarbonDioxide

carbondioxide→nextCarbonDioxide()

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

`YCarbonDioxide * nextCarbonDioxide()`

Returns :

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

carbondioxide→registerTimedReportCallback()**YCarbonDioxide****carbondioxide→****registerTimedReportCallback()**

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

```
int registerTimedReportCallback( YCarbonDioxideTimedReportCallback callback)
```

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

Parameters :

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

carbondioxide→registerValueCallback()**YCarbonDioxide****carbondioxide→registerValueCallback()**

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

```
int registerValueCallback( YCarbonDioxideValueCallback callback)
```

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

Parameters :

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

carbondioxide→**set_highestValue()****YCarbonDioxide****carbondioxide**→**setHighestValue()****carbondioxide**→**set_highestValue()**

Changes the recorded maximal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→**set_logFrequency()**

YCarbonDioxide

carbondioxide→**setLogFrequency()****carbondioxide**

→**set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logicalName()**YCarbonDioxide****carbondioxide→setLogicalName()**
carbondioxide→set_logicalName()

Changes the logical name of the CO2 sensor.

```
int set_logicalName( const string& newval)
```

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

Parameters :

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

Returns :

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

carbondioxide→set_lowestValue()

YCarbonDioxide

carbondioxide→setLowestValue()**carbondioxide→**
set_lowestValue()

Changes the recorded minimal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_reportFrequency()**YCarbonDioxide****carbondioxide→setReportFrequency()****carbondioxide→set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_resolution()

YCarbonDioxide

carbondioxide→setResolution()
carbondioxide→set_resolution()

Changes the resolution of the measured physical values.

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_userdata()**YCarbonDioxide****carbondioxide→setUserData()****carbondioxide→**
set_userdata()

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

```
void set_userdata( void* data)
```

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

Parameters :

data any kind of object to be stored

3.5. ColorLed function interface

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

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

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

Global functions

yFindColorLed(func)

Retrieves an RGB led for a given identifier.

yFirstColorLed()

Starts the enumeration of RGB leds currently accessible.

YColorLed methods

colorled→describe()

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

colorled→get_advertisedValue()

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

colorled→get_errorMessage()

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

colorled→get_errorType()

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

colorled→get_friendlyName()

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

colorled→get_functionDescriptor()

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

colorled→get_functionId()

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

colorled→get_hardwareId()

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

colorled→get_hslColor()

Returns the current HSL color of the led.

colorled→get_logicalName()

Returns the logical name of the RGB led.

colored→get_module()

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

colored→get_module_async(callback, context)

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

colored→get_rgbColor()

Returns the current RGB color of the led.

colored→get_rgbColorAtPowerOn()

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

colored→get_userData()

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

colored→hslMove(hsl_target, ms_duration)

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

colored→isOnline()

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

colored→isOnline_async(callback, context)

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

colored→load(msValidity)

Preloads the RGB led cache with a specified validity duration.

colored→load_async(msValidity, callback, context)

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

colored→nextColorLed()

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

colored→registerValueCallback(callback)

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

colored→rgbMove(rgb_target, ms_duration)

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

colored→set_hslColor(newval)

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

colored→set_logicalName(newval)

Changes the logical name of the RGB led.

colored→set_rgbColor(newval)

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

colored→set_rgbColorAtPowerOn(newval)

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

colored→set_userData(data)

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

colored→wait_async(callback, context)

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

YColorLed.FindColorLed()**YColorLed****yFindColorLed()**`yFindColorLed()`

Retrieves an RGB led for a given identifier.

```
YColorLed* yFindColorLed( const 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()
yFirstColorLed()**yFirstColorLed()**

YColorLed

Starts the enumeration of RGB leds currently accessible.

`YColorLed* yFirstColorLed()`

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()**YColorLed**

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

string **describe()**

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

Returns :

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

colorled→**get_advertisedValue()****YColorLed****colorled**→**advertisedValue()****colorled**→
get_advertisedValue()

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

string **get_advertisedValue()**

Returns :

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

colorled→**get_errorMessage()**

YColorLed

colorled→**errorMessage()****colorled**→
get_errorMessage()

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

string **get_errorMessage()**

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

Returns :

a string corresponding to the latest error message that occurred while using the RGB led object

colorled→**get_errorType()****YColorLed****colorled**→**errorType()****colorled**→**get_errorType()**

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

YRETCODE **get_errorType()**

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

Returns :

a number corresponding to the code of the latest error that occurred while using the RGB led object

colorled→**get_friendlyName()****YColorLed****colorled**→**friendlyName()****colorled**→**get_friendlyName()**

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

`string get_friendlyName()`

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

Returns :

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

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

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

YFUN_DESCR **get_functionDescriptor()**

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

Returns :

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

colorled→**get_functionId()**

YColorLed

colorled→**functionId()****colorled**→
get_functionId()

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

string **get_functionId()** ()

For example `relay1`

Returns :

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

colorled→**get_hardwareId()****YColorLed****colorled**→**hardwareId()****colorled**→
get_hardwareId()

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

`string` **get_hardwareId()**

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

Returns :

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

colorled→get_hslColor()

YColorLed

colorled→hslColor()**colorled→get_hslColor()**

Returns the current HSL color of the led.

int get_hslColor()

Returns :

an integer corresponding to the current HSL color of the led

On failure, throws an exception or returns Y_HSLCOLOR_INVALID.

colorled→**get_logicalName()****YColorLed****colorled**→**logicalName()****colorled**→
get_logicalName()

Returns the logical name of the RGB led.

string **get_logicalName()**

Returns :

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

colorled→get_module()

YColorLed

colorled→module()**colorled→get_module()**

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

YModule * **get_module()**

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

Returns :

an instance of YModule

colorled→get_rgbColor()**YColorLed****colorled→rgbColor()****colorled→get_rgbColor()**

Returns the current RGB color of the led.

int **get_rgbColor()**

Returns :

an integer corresponding to the current RGB color of the led

On failure, throws an exception or returns Y_RGBCOLOR_INVALID.

colorled→get_rgbColorAtPowerOn()

YColorLed

colorled→rgbColorAtPowerOn()colorled→

get_rgbColorAtPowerOn()

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

int get_rgbColorAtPowerOn()

Returns :

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

On failure, throws an exception or returns Y_RGBCOLORATPOWERON_INVALID.

colorled→get_userdata()**YColorLed****colorled→userdata()****colorled→get_userdata()**

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

```
void * get_userdata( )
```

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

Returns :

the object stored previously by the caller.

colorled→hslMove()**colorled→hslMove ()****YColorLed**

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

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

Parameters :

hsl_target desired HSL color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→isOnline()**colorled→isOnline()****YColorLed**

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

```
bool isOnline( )
```

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

Returns :

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

colorled→load()**colorled→load()****YColorLed**

Preloads the RGB led cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

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

Parameters :

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

Returns :

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

colorled→**nextColorLed()****colorled**→
nextColorLed()

YColorLed

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()****colorled**→
registerValueCallback()

YColorLed

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

```
int registerValueCallback( YColorLedValueCallback callback)
```

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

Parameters :

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

colorled→rgbMove()**colorled→rgbMove()****YColorLed**

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

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

Parameters :

rgb_target desired RGB color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_hslColor()**

YColorLed

colorled→**setHslColor()****colorled**→**set_hslColor()**

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

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

Encoding is done as follows: 0xHHSSL.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_logicalName()****YColorLed****colorled**→**setLogicalName()****colorled**→
set_logicalName()

Changes the logical name of the RGB led.

```
int set_logicalName( const string& newval)
```

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

Parameters :

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

Returns :

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

colorled→set_rgbColor()

YColorLed

colorled→setRgbColor()colorled→

set_rgbColor()

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

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

Encoding is done as follows: 0xRRGGBB.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_rgbColorAtPowerOn()****YColorLed****colorled**→**setRgbColorAtPowerOn()****colorled**→**set_rgbColorAtPowerOn()**

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

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→set_userdata()

YColorLed

colorled→setUserData()**colorled→**

set_userdata()

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

```
void set_userdata( void* data)
```

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

Parameters :

data any kind of object to be stored

3.6. Compass function interface

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

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

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

Global functions

yFindCompass(func)

Retrieves a compass for a given identifier.

yFirstCompass()

Starts the enumeration of compasses currently accessible.

YCompass methods

compass→calibrateFromPoints(rawValues, refValues)

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

compass→describe()

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

compass→get_advertisedValue()

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

compass→get_currentRawValue()

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

compass→get_currentValue()

Returns the current value of the relative bearing.

compass→get_errorMessage()

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

compass→get_errorType()

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

compass→get_friendlyName()

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

compass→get_functionDescriptor()

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

compass→get_functionId()

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

compass→get_hardwareId()

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

compass→get_highestValue()

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

compass→get_logFrequency()

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

compass→get_logicalName()

Returns the logical name of the compass.

compass→get_lowestValue()

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

compass→get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

compass→get_module()

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

compass→get_module_async(callback, context)

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

compass→get_recordedData(startTime, endTime)

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

compass→get_reportFrequency()

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

compass→get_resolution()

Returns the resolution of the measured values.

compass→get_unit()

Returns the measuring unit for the relative bearing.

compass→get_userData()

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

compass→isOnline()

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

compass→isOnline_async(callback, context)

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

compass→load(msValidity)

Preloads the compass cache with a specified validity duration.

compass→loadCalibrationPoints(rawValues, refValues)

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

compass→load_async(msValidity, callback, context)

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

compass→nextCompass()

Continues the enumeration of compasses started using yFirstCompass().

compass→registerTimedReportCallback(callback)

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

compass→registerValueCallback(callback)

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

compass→set_highestValue(newval)

Changes the recorded maximal value observed.

compass→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

compass→set_logicalName(newval)

Changes the logical name of the compass.

compass→set_lowestValue(newval)

Changes the recorded minimal value observed.

compass→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

compass→set_resolution(newval)

Changes the resolution of the measured physical values.

compass→set_userData(data)

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

compass→wait_async(callback, context)

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

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

Retrieves a compass for a given identifier.

```
YCompass* yFindCompass( const string& func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the compass

Returns :

a `YCompass` object allowing you to drive the compass.

YCompass.FirstCompass()
yFirstCompass()**yFirstCompass()**

YCompass

Starts the enumeration of compasses currently accessible.

YCompass* yFirstCompass()

Use the method `YCompass.nextCompass()` to iterate on next compasses.

Returns :

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

compass→**calibrateFromPoints()****compass**→
calibrateFromPoints()

YCompass

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

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

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→describe()**compass→describe()****YCompass**

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

string **describe()**

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

Returns :

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

compass→**get_advertisedValue()**

YCompass

compass→**advertisedValue()****compass**→

get_advertisedValue()

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

string **get_advertisedValue()**

Returns :

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

compass→**get_currentRawValue()****YCompass****compass**→**currentRawValue()****compass**→**get_currentRawValue()**

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

double **get_currentRawValue()**

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

compass→**get_currentValue()**

YCompass

compass→**currentValue()****compass**→

get_currentValue()

Returns the current value of the relative bearing.

`double get_currentValue()`

Returns :

a floating point number corresponding to the current value of the relative bearing

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

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

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

`string` **get_errorMessage()**

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

Returns :

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

compass→**get_errorType()**

YCompass

compass→**errorType()****compass**→**get_errorType()**

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

YRETCODE **get_errorType()**

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

Returns :

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

compass→**get_friendlyName()**
compass→**friendlyName()****compass**→
get_friendlyName()

YCompass

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

`string get_friendlyName()`

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

Returns :

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

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

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

YFUN_DESCR **get_functionDescriptor()**

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

Returns :

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

compass→**get_functionId()****YCompass****compass**→**functionId()****compass**→
get_functionId()

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

string **get_functionId()**

For example `relay1`

Returns :

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

compass→**get_hardwareId()**

YCompass

compass→**hardwareId()****compass**→
get_hardwareId()

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

`string get_hardwareId()`

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

Returns :

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

compass→**get_highestValue()****YCompass****compass**→**highestValue()****compass**→
get_highestValue()

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

double **get_highestValue()**

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

compass→**get_logFrequency()****YCompass****compass**→**logFrequency()****compass**→**get_logFrequency()**

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

string **get_logFrequency()**

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

compass→**get_logicalName()**
compass→**logicalName()****compass**→
get_logicalName()

YCompass

Returns the logical name of the compass.

string **get_logicalName()**

Returns :

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

compass→**get_lowestValue()**

YCompass

compass→**lowestValue()****compass**→
get_lowestValue()

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

double **get_lowestValue()**

Returns :

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

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

compass→**get_magneticHeading()****YCompass****compass**→**magneticHeading()****compass**→
get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

double **get_magneticHeading()**

Returns :

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

On failure, throws an exception or returns Y_MAGNETICHEADING_INVALID.

compass→get_module()

YCompass

compass→module()**compass→get_module()**

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

YModule * **get_module()**

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

Returns :

an instance of YModule

compass→**get_recordedData()**
compass→**recordedData()****compass**→
get_recordedData()

YCompass

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

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

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

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

Parameters :

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

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

Returns :

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

compass→**get_reportFrequency()****YCompass****compass**→**reportFrequency()****compass**→**get_reportFrequency()**

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

string **get_reportFrequency()****Returns :**

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

compass→**get_resolution()****YCompass****compass**→**resolution()****compass**→
get_resolution()

Returns the resolution of the measured values.

`double` **get_resolution()** `()`

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

Returns :

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

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

compass→**get_unit()**

YCompass

compass→**unit()****compass**→**get_unit()**

Returns the measuring unit for the relative bearing.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the relative bearing

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

compass→get_userData()**YCompass****compass→userData()****compass→get_userData()**

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

```
void * get_userData( )
```

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

Returns :

the object stored previously by the caller.

compass→**isOnline()****compass**→**isOnline()**

YCompass

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

`bool isOnline()`

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

Returns :

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

compass→load()**compass→load()****YCompass**

Preloads the compass cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

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

Parameters :

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

Returns :

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

compass→**loadCalibrationPoints()****compass**→
loadCalibrationPoints()

YCompass

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

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**nextCompass()****compass**→
nextCompass ()

YCompass

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

`YCompass *` **nextCompass ()**

Returns :

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

compass→**registerTimedReportCallback()****compass**→
registerTimedReportCallback()**YCompass**

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

```
int registerTimedReportCallback( YCompassTimedReportCallback callback)
```

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

Parameters :

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

compass→**registerValueCallback()****compass**→
registerValueCallback()

YCompass

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

```
int registerValueCallback( YCompassValueCallback callback)
```

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

Parameters :

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

compass→**set_highestValue()****YCompass****compass**→**setHighestValue()****compass**→
set_highestValue()

Changes the recorded maximal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_logFrequency()****YCompass****compass**→**setLogFrequency()****compass**→
set_logFrequency()

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_logicalName()****YCompass****compass**→**setLogicalName()****compass**→**set_logicalName()**

Changes the logical name of the compass.

```
int set_logicalName( const string& newval)
```

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

Parameters :

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

Returns :

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

compass→**set_lowestValue()****YCompass****compass**→**setLowestValue()****compass**→
set_lowestValue()

Changes the recorded minimal value observed.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_reportFrequency()****YCompass****compass**→**setReportFrequency()****compass**→
set_reportFrequency()

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_resolution()****YCompass****compass**→**setResolution()****compass**→
set_resolution()

Changes the resolution of the measured physical values.

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set_userdata()

YCompass

compass→set_userdata()
compass→set_userdata()

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

```
void set_userdata( void* data)
```

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

Parameters :

data any kind of object to be stored

3.7. Current function interface

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

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

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

Global functions

yFindCurrent(func)

Retrieves a current sensor for a given identifier.

yFirstCurrent()

Starts the enumeration of current sensors currently accessible.

YCurrent methods

current→calibrateFromPoints(rawValues, refValues)

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

current→describe()

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

current→get_advertisedValue()

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

current→get_currentRawValue()

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

current→get_currentValue()

Returns the current measure for the current.

current→get_errorMessage()

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

current→get_errorType()

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

current→get_friendlyName()

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

current→get_functionDescriptor()

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

current→get_functionId()

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

current→get_hardwareId()

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

current→get_highestValue()

Returns the maximal value observed for the current.

current→get_logFrequency()

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

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

Returns the minimal value observed for the current.

current→get_module()

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

current→get_module_async(callback, context)

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

current→get_recordedData(startTime, endTime)

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

current→get_reportFrequency()

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

current→get_resolution()

Returns the resolution of the measured values.

current→get_unit()

Returns the measuring unit for the current.

current→get_userData()

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

current→isOnline()

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

current→isOnline_async(callback, context)

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

current→load(msValidity)

Preloads the current sensor cache with a specified validity duration.

current→loadCalibrationPoints(rawValues, refValues)

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

current→load_async(msValidity, callback, context)

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

current→nextCurrent()

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

current→registerTimedReportCallback(callback)

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

current→registerValueCallback(callback)

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

current→set_highestValue(newval)

Changes the recorded maximal value observed pour the current.

current→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

current→set_logicalName(newval)

Changes the logical name of the current sensor.

current→**set_lowestValue**(newval)

Changes the recorded minimal value observed pour the current.

current→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

current→**set_resolution**(newval)

Changes the resolution of the measured values.

current→**set_userData**(data)

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

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

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

YCurrent.FindCurrent() yFindCurrent()yFindCurrent ()

YCurrent

Retrieves a current sensor for a given identifier.

```
YCurrent* yFindCurrent( const string& func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the current sensor

Returns :

a `YCurrent` object allowing you to drive the current sensor.

YCurrent.FirstCurrent()**YCurrent****yFirstCurrent()****yFirstCurrent()**

Starts the enumeration of current sensors currently accessible.

YCurrent* **yFirstCurrent()**

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()****current**→
calibrateFromPoints()

YCurrent

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

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

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→describe()**current→describe()****YCurrent**

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

string **describe()**

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

Returns :

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

current→**get_advertisedValue()**

YCurrent

current→**advertisedValue()****current**→
get_advertisedValue()

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

string **get_advertisedValue()**

Returns :

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

current→**get_currentRawValue()****YCurrent****current**→**currentRawValue()****current**→
get_currentRawValue()

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

double **get_currentRawValue()**

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

current→**get_currentValue()**

YCurrent

current→**currentValue()****current**→

get_currentValue()

Returns the current measure for the current.

`double get_currentValue()`

Returns :

a floating point number corresponding to the current measure for the current

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

current→**get_errorMessage()****YCurrent****current**→**errorMessage()****current**→
get_errorMessage()

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

`string` **get_errorMessage()**

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

Returns :

a string corresponding to the latest error message that occurred while using the current sensor object

current→**get_errorType()**

YCurrent

current→**errorType()****current**→**get_errorType()**

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

YRETCODE **get_errorType()**

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

Returns :

a number corresponding to the code of the latest error that occurred while using the current sensor object

current→**get_friendlyName()**
current→**friendlyName()****current**→
get_friendlyName()

YCurrent

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

`string` **get_friendlyName()**

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

Returns :

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

current→**get_functionDescriptor()**

YCurrent

current→**functionDescriptor()****current**→
get_functionDescriptor()

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

YFUN_DESCR **get_functionDescriptor()**

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

Returns :

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

current→get_functionId()**YCurrent****current→functionId()****current→get_functionId()**

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

string **get_functionId()**

For example `relay1`

Returns :

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

current→get_hardwareId()

YCurrent

current→hardwareId()**current→get_hardwareId()**

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

string **get_hardwareId()**

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

Returns :

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

current→**get_highestValue()**
current→**highestValue()****current**→
get_highestValue()

YCurrent

Returns the maximal value observed for the current.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the current

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

current→**get_logFrequency()**

YCurrent

current→**logFrequency()****current**→
get_logFrequency()

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

string **get_logFrequency()**

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

current→**get_logicalName()**
current→**logicalName()****current**→
get_logicalName()

YCurrent

Returns the logical name of the current sensor.

string **get_logicalName()**

Returns :

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

current→**get_lowestValue()**

YCurrent

current→**lowestValue()****current**→
get_lowestValue()

Returns the minimal value observed for the current.

`double get_lowestValue()`

Returns :

a floating point number corresponding to the minimal value observed for the current

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

current→get_module()**YCurrent****current→module()****current→get_module()**

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

YModule * **get_module()**

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

Returns :

an instance of YModule

current→**get_recordedData()****YCurrent****current**→**recordedData()****current**→**get_recordedData()**

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

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

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

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

Parameters :

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

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

Returns :

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

current→get_reportFrequency()**YCurrent****current→reportFrequency()****current→****get_reportFrequency()**

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

string **get_reportFrequency()**

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

current→get_resolution()

YCurrent

current→resolution()**current→get_resolution()**

Returns the resolution of the measured values.

double **get_resolution()** ()

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

current→**get_unit()****YCurrent****current**→**unit()****current**→**get_unit()**

Returns the measuring unit for the current.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the current

On failure, throws an exception or returns Y_UNIT_INVALID.

current→get_userdata()

YCurrent

current→userData()**current→get_userdata()**

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

`void * get_userdata()`

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

Returns :

the object stored previously by the caller.

current→isOnline()**current→isOnline()****YCurrent**

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

```
bool isOnline( )
```

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

Returns :

`true` if the current sensor can be reached, and `false` otherwise

current→load()**current→load()****YCurrent**

Preloads the current sensor cache with a specified validity duration.

YRETCODE **load(int msValidity)**

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

Parameters :

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

Returns :

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

current→**loadCalibrationPoints()****current**→
loadCalibrationPoints()

YCurrent

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

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**nextCurrent()****current**→**nextCurrent ()**

YCurrent

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

`YCurrent * nextCurrent()`

Returns :

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

current→**registerTimedReportCallback()****current**→
registerTimedReportCallback()

YCurrent

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

```
int registerTimedReportCallback( YCurrentTimedReportCallback callback)
```

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

Parameters :

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

current→**registerValueCallback()****current**→
registerValueCallback()

YCurrent

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

```
int registerValueCallback( YCurrentValueCallback callback)
```

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

Parameters :

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

current→**set_highestValue()**
current→**setHighestValue()****current**→
set_highestValue()

YCurrent

Changes the recorded maximal value observed pour the current.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_logFrequency()****YCurrent****current**→**setLogFrequency()****current**→**set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_logicalName()****YCurrent****current**→**setLogicalName()****current**→
set_logicalName()

Changes the logical name of the current sensor.

```
int set_logicalName( const string& newval)
```

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

Parameters :

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

Returns :

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

current→**set_lowestValue()****YCurrent****current**→**setLowestValue()****current**→**set_lowestValue()**

Changes the recorded minimal value observed pour the current.

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_reportFrequency()****YCurrent****current**→**setReportFrequency()****current**→**set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_resolution()****YCurrent****current**→**setResolution()****current**→
set_resolution()

Changes the resolution of the measured values.

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

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→set_userdata()**YCurrent****current→setUserData()****current→set_userdata()**

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

```
void set_userdata( void* data)
```

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

Parameters :

data any kind of object to be stored

3.8. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

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

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

Global functions

yFindDataLogger(func)

Retrieves a data logger for a given identifier.

yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

YDataLogger methods

datalogger→describe()

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

datalogger→forgetAllDataStreams()

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

datalogger→get_advertisedValue()

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

datalogger→get_autoStart()

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

datalogger→get_currentRunIndex()

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

datalogger→get_dataSets()

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

datalogger→get_dataStreams(v)

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

datalogger→get_errorMessage()

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

datalogger→get_errorType()

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

datalogger→get_friendlyName()

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

datalogger→get_functionDescriptor()

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

`datalogger→get_functionId()`

Returns the hardware identifier of the data logger, without reference to the module.

`datalogger→get_hardwareId()`

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

`datalogger→get_logicalName()`

Returns the logical name of the data logger.

`datalogger→get_module()`

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

`datalogger→get_module_async(callback, context)`

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

`datalogger→get_recording()`

Returns the current activation state of the data logger.

`datalogger→get_timeUTC()`

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

`datalogger→get_userData()`

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

`datalogger→isOnline()`

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

`datalogger→isOnline_async(callback, context)`

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

`datalogger→load(msValidity)`

Preloads the data logger cache with a specified validity duration.

`datalogger→load_async(msValidity, callback, context)`

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

`datalogger→nextDataLogger()`

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

`datalogger→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`datalogger→set_autoStart(newval)`

Changes the default activation state of the data logger on power up.

`datalogger→set_logicalName(newval)`

Changes the logical name of the data logger.

`datalogger→set_recording(newval)`

Changes the activation state of the data logger to start/stop recording data.

`datalogger→set_timeUTC(newval)`

Changes the current UTC time reference used for recorded data.

`datalogger→set_userData(data)`

Stores a user context provided as argument in the `userData` attribute of the function.

`datalogger→wait_async(callback, context)`

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDataLogger.FindDataLogger()**YDataLogger****yFindDataLogger()****yFindDataLogger()**

Retrieves a data logger for a given identifier.

```
YDataLogger* yFindDataLogger( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger()**YDataLogger****yFirstDataLogger()****yFirstDataLogger()**

Starts the enumeration of data loggers currently accessible.

`YDataLogger* yFirstDataLogger()`

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

datalogger→describe()**datalogger→describe()****YDataLogger**

Returns a short text that describes unambiguously the instance of the data logger in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the data logger (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

dataLogger→forgetAllDataStreams()dataLogger→
forgetAllDataStreams()

YDataLogger

Clears the data logger memory and discards all recorded data streams.

int forgetAllDataStreams()

This method also resets the current run index to zero.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→**get_advertisedValue()**

YDataLogger

datalogger→**advertisedValue()****datalogger**→

get_advertisedValue()

Returns the current value of the data logger (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the data logger (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

dataLogger→get_autoStart()**YDataLogger****dataLogger→autoStart()****dataLogger→****get_autoStart()**

Returns the default activation state of the data logger on power up.

[Y_AUTOSTART_enum](#) **get_autoStart()**

Returns :

either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the default activation state of the data logger on power up

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

dataLogger→**get_currentRunIndex()****YDataLogger****dataLogger**→**currentRunIndex()****dataLogger**→**get_currentRunIndex()**

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

int **get_currentRunIndex()****Returns :**

an integer corresponding to the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point

On failure, throws an exception or returns Y_CURRENTRUNINDEX_INVALID.

datalogger→**get_dataSets()****YDataLogger****datalogger**→**dataSets()****datalogger**→
get_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

```
vector<YDataSet> get_dataSets()
```

This function only works if the device uses a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

Returns :

a list of YDataSet object.

On failure, throws an exception or returns an empty list.

datalogger→**get_dataStreams()****YDataLogger****datalogger**→**dataStreams()****datalogger**→
get_dataStreams()

Builds a list of all data streams hold by the data logger (legacy method).

```
int get_dataStreams( )
```

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

▼ an array of YDataStream objects to be filled in

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→get_errorMessage()**YDataLogger****datalogger→errorMessage()****datalogger→**
get_errorMessage()

Returns the error message of the latest error with the data logger.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the data logger object

datalogger→**get_errorType()**

YDataLogger

datalogger→**errorType()****datalogger**→

get_errorType()

Returns the numerical error code of the latest error with the data logger.

[YRETCODE](#) **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the data logger object

dataLogger→get_friendlyName()**YDataLogger****dataLogger→friendlyName()****dataLogger→****get_friendlyName()**

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the data logger using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

datalogger→**get_functionDescriptor()**

YDataLogger

datalogger→**functionDescriptor()****datalogger**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

datalogger→**get_functionId()****YDataLogger****datalogger**→**functionId()****datalogger**→
get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

`string` **get_functionId()**

For example `relay1`

Returns :

a string that identifies the data logger (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

datalogger→**get_hardwareId()**

YDataLogger

datalogger→**hardwareId()****datalogger**→
get_hardwareId()

Returns the unique hardware identifier of the data logger in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the data logger (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

datalogger→**get_logicalName()****YDataLogger****datalogger**→**logicalName()****datalogger**→
get_logicalName()

Returns the logical name of the data logger.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the data logger. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

datalogger→get_module()

YDataLogger

datalogger→module()**datalogger→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

datalogger→get_recording()
datalogger→recording()**datalogger→**
get_recording()

YDataLogger

Returns the current activation state of the data logger.

[Y_RECORDING_enum](#) **get_recording()**

Returns :

either Y_RECORDING_OFF or Y_RECORDING_ON, according to the current activation state of the data logger

On failure, throws an exception or returns Y_RECORDING_INVALID.

datalogger→**get_timeUTC()**

YDataLogger

datalogger→**timeUTC()****datalogger**→
get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

s64 **get_timeUTC()**

Returns :

an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns Y_TIMEUTC_INVALID.

dataLogger→get_userData()**YDataLogger****dataLogger→userData()****dataLogger→****get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
void * get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

datalogger→isOnline()**datalogger→isOnline()**

YDataLogger

Checks if the data logger is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

Returns :

`true` if the data logger can be reached, and `false` otherwise

dataLogger→load()**dataLogger→load()****YDataLogger**

Preloads the data logger cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

dataLogger→**nextDataLogger()****dataLogger**→
nextDataLogger ()

YDataLogger

Continues the enumeration of data loggers started using `yFirstDataLogger ()`.

`YDataLogger * nextDataLogger ()`

Returns :

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

datalogger→registerValueCallback()datalogger→
registerValueCallback()

YDataLogger

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YDataLoggerValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

datalogger→**set_autoStart()****YDataLogger****datalogger**→**setAutoStart()****datalogger**→
set_autoStart()

Changes the default activation state of the data logger on power up.

```
int set_autoStart( Y_AUTOSTART_enum newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the default activation state of the data logger on power up

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→**set_logicalName()****YDataLogger****datalogger**→**setLogicalName()****datalogger**→**set_logicalName()**

Changes the logical name of the data logger.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the data logger.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

datalogger→**set_recording()****YDataLogger****datalogger**→**setRecording()****datalogger**→
set_recording()

Changes the activation state of the data logger to start/stop recording data.

```
int set_recording( Y_RECORDING_enum newval)
```

Parameters :

newval either Y_RECORDING_OFF or Y_RECORDING_ON, according to the activation state of the data logger to start/stop recording data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_timeUTC()**YDataLogger****datalogger→setTimeUTC()**
datalogger→set_timeUTC()

Changes the current UTC time reference used for recorded data.

```
int set_timeUTC( s64 newval)
```

Parameters :

newval an integer corresponding to the current UTC time reference used for recorded data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_userdata()

YDataLogger

datalogger→setUserData()**datalogger→**

set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.9. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_datalogger.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDataLogger = yoctolib.YDataLogger;
php	require_once('yocto_datalogger.php');
c++	#include "yocto_datalogger.h"
m	#import "yocto_datalogger.h"
pas	uses yocto_datalogger;
vb	yocto_datalogger.vb
cs	yocto_datalogger.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_datalogger import *

YDataRun methods
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_startTimeUTC()

YDataRun

datarun→startTimeUTC()

Returns the start time of the data run, relative to the Jan 1, 1970.

If the UTC time was not set in the datalogger at any time during the recording of this data run, and if this is not the current run, this method returns 0.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data run (i.e. Unix time representation of the absolute time).

3.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instantiated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YAPI = yoctolib.YAPI;</code> <code>var YModule = yoctolib.YModule;</code>
php	<code>require_once('yocto_api.php');</code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>

YDataSet methods

dataset→`get_endTimeUTC()`

Returns the end time of the dataset, relative to the Jan 1, 1970.

dataset→`get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

dataset→`get_hardwareId()`

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

dataset→`get_measures()`

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

dataset→`get_preview()`

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

dataset→`get_progress()`

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dataset→`get_startTimeUTC()`

Returns the start time of the dataset, relative to the Jan 1, 1970.

dataset→`get_summary()`

Returns an YMeasure object which summarizes the whole DataSet.

dataset→`get_unit()`

Returns the measuring unit for the measured value.

3. Reference

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→**get_endTimeUTC()**
dataset→**endTimeUTC()****dataset**→
get_endTimeUTC()

YDataSet

Returns the end time of the dataset, relative to the Jan 1, 1970.

s64 **get_endTimeUTC()**

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the `dataLogger` within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

dataset→get_functionId()

YDataSet

dataset→functionId()**dataset→get_functionId()**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

string **get_functionId()** ()

For example `temperature1`.

Returns :

a string that identifies the function (ex: `temperature1`)

dataset→**get_hardwareId()****YDataSet****dataset**→**hardwareId()****dataset**→**get_hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `THRMCPL1-123456.temperature1`)

Returns :

a string that uniquely identifies the function (ex: `THRMCPL1-123456.temperature1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

dataset→**get_measures()****YDataSet****dataset**→**measures()****dataset**→**get_measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

```
vector<YMeasure> get_measures( )
```

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :

a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→**get_preview()****YDataSet****dataset**→**preview()****dataset**→**get_preview()**

Returns a condensed version of the measures that can be retrieved in this YDataSet, as a list of YMeasure objects.

```
vector<YMeasure> get_preview()
```

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_progress()**YDataSet****dataset→progress()****dataset→get_progress()**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

int get_progress()

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :

an integer in the range 0 to 100 (percentage of completion).

dataset→**get_startTimeUTC()****YDataSet****dataset**→**startTimeUTC()****dataset**→
get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

s64 **get_startTimeUTC()**

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

dataset→**get_summary()**

YDataSet

dataset→**summary()****dataset**→**get_summary()**

Returns an YMeasure object which summarizes the whole DataSet.

YMeasure **get_summary()**

It includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

Returns :

an YMeasure object

dataset→**get_unit()****YDataSet****dataset**→**unit()****dataset**→**get_unit()**

Returns the measuring unit for the measured value.

string **get_unit()**

Returns :

a string that represents a physical unit.

On failure, throws an exception or returns Y_UNIT_INVALID.

dataset→**loadMore()****dataset**→**loadMore()**

YDataSet

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

int loadMore()

Returns :

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

3.11. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
c++	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

YDataStream methods
datastream→get_averageValue() Returns the average of all measures observed within this stream.
datastream→get_columnCount() Returns the number of data columns present in this stream.
datastream→get_columnNames() Returns the title (or meaning) of each data column present in this stream.
datastream→get_data(row, col) Returns a single measure from the data stream, specified by its row and column index.
datastream→get_dataRows() Returns the whole data set contained in the stream, as a bidimensional table of numbers.
datastream→get_dataSamplesIntervalMs() Returns the number of milliseconds between two consecutive rows of this data stream.
datastream→get_duration() Returns the approximate duration of this stream, in seconds.
datastream→get_maxValue() Returns the largest measure observed within this stream.
datastream→get_minValue() Returns the smallest measure observed within this stream.
datastream→get_rowCount() Returns the number of data rows present in this stream.
datastream→get_runIndex() Returns the run index of the data stream.
datastream→get_startTime() Returns the relative start time of the data stream, measured in seconds.
datastream→get_startTimeUTC()

3. Reference

Returns the start time of the data stream, relative to the Jan 1, 1970.

datastream→**get_averageValue()****YDataStream****datastream**→**averageValue()****datastream**→
get_averageValue()

Returns the average of all measures observed within this stream.

`double` **get_averageValue()**

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→**get_columnCount()**

YDataStream

datastream→**columnCount()****datastream**→

get_columnCount()

Returns the number of data columns present in this stream.

```
int get_columnCount( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

datastream→**get_columnNames()****YDataStream****datastream**→**columnNames()****datastream**→
get_columnNames()

Returns the title (or meaning) of each data column present in this stream.

```
vector<string> get_columnNames( )
```

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes `_min`, `_avg` and `_max` respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

a list containing as many strings as there are columns in the data stream.

On failure, throws an exception or returns an empty array.

datastream→get_data()

YDataStream

datastream→data()**datastream→get_data()**

Returns a single measure from the data stream, specified by its row and column index.

```
double get_data( int row, int col)
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Parameters :

row row index

col column index

Returns :

a floating-point number

On failure, throws an exception or returns `Y_DATA_INVALID`.

datastream→get_dataRows()**YDataStream****datastream→dataRows()****datastream→**
get_dataRows()

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

```
vector< vector<double> > get_dataRows( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Returns :

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.

datastream→**get_dataSamplesIntervalMs()** **YDataStream**
datastream→**dataSamplesIntervalMs()****datastream**→
get_dataSamplesIntervalMs()

Returns the number of milliseconds between two consecutive rows of this data stream.

```
int get_dataSamplesIntervalMs( )
```

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

Returns :

an unsigned number corresponding to a number of milliseconds.

datastream→**get_duration()****YDataStream****datastream**→**duration()****datastream**→
get_duration()

Returns the approximate duration of this stream, in seconds.

int **get_duration()**

Returns :

the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

datastream→**get_maxValue()**

YDataStream

datastream→**maxValue()****datastream**→

get_maxValue()

Returns the largest measure observed within this stream.

```
double get_maxValue()
```

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→**get_minValue()****YDataStream****datastream**→**minValue()****datastream**→**get_minValue()**

Returns the smallest measure observed within this stream.

```
double get_minValue()
```

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the smallest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_rowCount()

YDataStream

datastream→rowCount()**datastream→**

get_rowCount ()

Returns the number of data rows present in this stream.

int get_rowCount()

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

datastream→**get_runIndex()****YDataStream****datastream**→**runIndex()****datastream**→
get_runIndex()

Returns the run index of the data stream.

int **get_runIndex()**

A run can be made of multiple datastreams, for different time intervals.

Returns :

an unsigned number corresponding to the run index.

datastream→**get_startTime()****YDataStream****datastream**→**startTime()****datastream**→**get_startTime()**

Returns the relative start time of the data stream, measured in seconds.

```
int get_startTime( )
```

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `get_startTimeUTC()`.

Returns :

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

datastream→**get_startTimeUTC()****YDataStream****datastream**→**startTimeUTC()****datastream**→
get_startTimeUTC()

Returns the start time of the data stream, relative to the Jan 1, 1970.

s64 **get_startTimeUTC()**

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data stream (i.e. Unix time representation of the absolute time).

3.12. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_digitalio.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDigitalIO = yoctolib.YDigitalIO;
php	require_once('yocto_digitalio.php');
c++	#include "yocto_digitalio.h"
m	#import "yocto_digitalio.h"
pas	uses yocto_digitalio;
vb	yocto_digitalio.vb
cs	yocto_digitalio.cs
java	import com.yoctopuce.YoctoAPI.YDigitalIO;
py	from yocto_digitalio import *

Global functions

yFindDigitalIO(func)

Retrieves a digital IO port for a given identifier.

yFirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

YDigitalIO methods

digitalio→delayedPulse(bitno, ms_delay, ms_duration)

Schedules a pulse on a single bit for a specified duration.

digitalio→describe()

Returns a short text that describes unambiguously the instance of the digital IO port in the form TYPE (NAME) = SERIAL . FUNCTIONID.

digitalio→get_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

digitalio→get_bitDirection(bitno)

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

digitalio→get_bitOpenDrain(bitno)

Returns the type of electrical interface of a single bit from the I/O port.

digitalio→get_bitPolarity(bitno)

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

digitalio→get_bitState(bitno)

Returns the state of a single bit of the I/O port.

digitalio→get_errorMessage()

Returns the error message of the latest error with the digital IO port.

digitalio→get_errorType()

Returns the numerical error code of the latest error with the digital IO port.

digitalio→get_friendlyName()

Returns a global identifier of the digital IO port in the format MODULE_NAME . FUNCTION_NAME.

digitalio→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

digitalio→get_functionId()

Returns the hardware identifier of the digital IO port, without reference to the module.

digitalio→get_hardwareId()

Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID.

digitalio→get_logicalName()

Returns the logical name of the digital IO port.

digitalio→get_module()

Gets the YModule object for the device on which the function is located.

digitalio→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

digitalio→get_outputVoltage()

Returns the voltage source used to drive output bits.

digitalio→get_portDirection()

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→get_portOpenDrain()

Returns the electrical interface for each bit of the port.

digitalio→get_portPolarity()

Returns the polarity of all the bits of the port.

digitalio→get_portSize()

Returns the number of bits implemented in the I/O port.

digitalio→get_portState()

Returns the digital IO port state: bit 0 represents input 0, and so on.

digitalio→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

digitalio→isOnline()

Checks if the digital IO port is currently reachable, without raising any error.

digitalio→isOnline_async(callback, context)

Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

digitalio→load(msValidity)

Preloads the digital IO port cache with a specified validity duration.

digitalio→load_async(msValidity, callback, context)

Preloads the digital IO port cache with a specified validity duration (asynchronous version).

digitalio→nextDigitalIO()

Continues the enumeration of digital IO ports started using yFirstDigitalIO().

digitalio→pulse(bitno, ms_duration)

Triggers a pulse on a single bit for a specified duration.

digitalio→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

digitalio→set_bitDirection(bitno, bitdirection)

Changes the direction of a single bit from the I/O port.

digitalio→set_bitOpenDrain(bitno, opendrain)

Changes the electrical interface of a single bit from the I/O port.

digitalio→set_bitPolarity(bitno, bitpolarity)

3. Reference

Changes the polarity of a single bit from the I/O port.

digitalio→**set_bitState**(**bitno**, **bitstate**)

Sets a single bit of the I/O port.

digitalio→**set_logicalName**(**newval**)

Changes the logical name of the digital IO port.

digitalio→**set_outputVoltage**(**newval**)

Changes the voltage source used to drive output bits.

digitalio→**set_portDirection**(**newval**)

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→**set_portOpenDrain**(**newval**)

Changes the electrical interface for each bit of the port.

digitalio→**set_portPolarity**(**newval**)

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→**set_portState**(**newval**)

Changes the digital IO port state: bit 0 represents input 0, and so on.

digitalio→**set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

digitalio→**toggle_bitState**(**bitno**)

Reverts a single bit of the I/O port.

digitalio→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDigitalIO.FindDigitalIO() yFindDigitalIO()yFindDigitalIO()

YDigitalIO

Retrieves a digital IO port for a given identifier.

```
YDigitalIO* yFindDigitalIO( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the digital IO port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDigitalIO.isOnline()` to test if the digital IO port is indeed online at a given time. In case of ambiguity when looking for a digital IO port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the digital IO port

Returns :

a `YDigitalIO` object allowing you to drive the digital IO port.

YDigitalIO.FirstDigitalIO()

YDigitalIO

yFirstDigitalIO()**yFirstDigitalIO()**

Starts the enumeration of digital IO ports currently accessible.

YDigitalIO* **yFirstDigitalIO()**

Use the method `YDigitalIO.nextDigitalIO()` to iterate on next digital IO ports.

Returns :

a pointer to a `YDigitalIO` object, corresponding to the first digital IO port currently online, or a `null` pointer if there are none.

digitalio→delayedPulse()**digitalio→**
delayedPulse()

YDigitalIO

Schedules a pulse on a single bit for a specified duration.

```
int delayedPulse( int bitno, int ms_delay, int ms_duration)
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno	the bit number; lowest bit has index 0
ms_delay	waiting time before the pulse, in milliseconds
ms_duration	desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→describe()**digitalio→describe()****YDigitalIO**

Returns a short text that describes unambiguously the instance of the digital IO port in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the digital IO port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

digitalio→**get_advertisedValue()****YDigitalIO****digitalio**→**advertisedValue()****digitalio**→**get_advertisedValue()**

Returns the current value of the digital IO port (no more than 6 characters).

`string` **get_advertisedValue()**

Returns :

a string corresponding to the current value of the digital IO port (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

digitalio→**get_bitDirection()****YDigitalIO****digitalio**→**bitDirection()****digitalio**→**get_bitDirection()**

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

```
int get_bitDirection( int bitno)
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**get_bitOpenDrain()**
digitalio→**bitOpenDrain()****digitalio**→
get_bitOpenDrain()

YDigitalIO

Returns the type of electrical interface of a single bit from the I/O port.

```
int get_bitOpenDrain( int bitno)
```

(0 means the bit is an input, 1 an output).

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

0 means the a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

On failure, throws an exception or returns a negative error code.

digitalio→**get_bitPolarity()****YDigitalIO****digitalio**→**bitPolarity()****digitalio**→**get_bitPolarity()**

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

```
int get_bitPolarity( int bitno)
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**get_bitState()****YDigitalIO****digitalio**→**bitState()****digitalio**→**get_bitState()**

Returns the state of a single bit of the I/O port.

```
int get_bitState( int bitno)
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

the bit state (0 or 1)

On failure, throws an exception or returns a negative error code.

digitalio→get_errorMessage()

YDigitalIO

digitalio→errorMessage()**digitalio→**

get_errorMessage()

Returns the error message of the latest error with the digital IO port.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the digital IO port object

```
digitalio→get_errorType()  
digitalio→errorType()digitalio→  
get_errorType( )
```

YDigitalIO

Returns the numerical error code of the latest error with the digital IO port.

YRETCODE `get_errorType()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the digital IO port object

digitalio→**get_friendlyName()****YDigitalIO****digitalio**→**friendlyName()****digitalio**→**get_friendlyName()**

Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the digital IO port if they are defined, otherwise the serial number of the module and the hardware identifier of the digital IO port (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the digital IO port using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

digitalio→**get_functionDescriptor()****YDigitalIO****digitalio**→**functionDescriptor()****digitalio**→
get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

`YFUN_DESCR` **get_functionDescriptor()**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

digitalio→**get_functionId()**

YDigitalIO

digitalio→**functionId()****digitalio**→
get_functionId()

Returns the hardware identifier of the digital IO port, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the digital IO port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

digitalio→**get_hardwareId()**
digitalio→**hardwareId()****digitalio**→
get_hardwareId()

YDigitalIO

Returns the unique hardware identifier of the digital IO port in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the digital IO port. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the digital IO port (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

digitalio→**get_logicalName()**

YDigitalIO

digitalio→**logicalName()****digitalio**→

get_logicalName()

Returns the logical name of the digital IO port.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the digital IO port. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

digitalio→get_module()**YDigitalIO****digitalio→module()****digitalio→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

digitalio→get_outputVoltage()**YDigitalIO****digitalio→outputVoltage()****digitalio→****get_outputVoltage()**

Returns the voltage source used to drive output bits.

Y_OUTPUTVOLTAGE_enum **get_outputVoltage()**

Returns :

a value among Y_OUTPUTVOLTAGE_USB_5V, Y_OUTPUTVOLTAGE_USB_3V and Y_OUTPUTVOLTAGE_EXT_V corresponding to the voltage source used to drive output bits

On failure, throws an exception or returns Y_OUTPUTVOLTAGE_INVALID.

digitalio→**get_portDirection()****YDigitalIO****digitalio**→**portDirection()****digitalio**→**get_portDirection()**

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

int **get_portDirection()**

Returns :

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

On failure, throws an exception or returns Y_PORTDIRECTION_INVALID.

digitalio→get_portOpenDrain()**YDigitalIO****digitalio→portOpenDrain()****digitalio→****get_portOpenDrain()**

Returns the electrical interface for each bit of the port.

int get_portOpenDrain()

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

Returns :

an integer corresponding to the electrical interface for each bit of the port

On failure, throws an exception or returns Y_PORTOPENDRAIN_INVALID.

digitalio→**get_portPolarity()****YDigitalIO****digitalio**→**portPolarity()****digitalio**→
get_portPolarity()

Returns the polarity of all the bits of the port.

int **get_portPolarity()** **()**

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

Returns :

an integer corresponding to the polarity of all the bits of the port

On failure, throws an exception or returns Y_PORTPOLARITY_INVALID.

digitalio→get_portSize()

YDigitalIO

digitalio→portSize()**digitalio→get_portSize()**

Returns the number of bits implemented in the I/O port.

```
int get_portSize( )
```

Returns :

an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns Y_PORTSIZE_INVALID.

digitalio→get_portState()**YDigitalIO****digitalio→portState()****digitalio→get_portState()**

Returns the digital IO port state: bit 0 represents input 0, and so on.

```
int get_portState( )
```

Returns :

an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

On failure, throws an exception or returns Y_PORTSTATE_INVALID.

digitalio→get_userdata()

YDigitalIO

digitalio→userdata()**digitalio→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

digitalio→isOnline()**digitalio→isOnline()****YDigitalIO**

Checks if the digital IO port is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the digital IO port.

Returns :

`true` if the digital IO port can be reached, and `false` otherwise

digitalio→load()**digitalio→load()****YDigitalIO**

Preloads the digital IO port cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

digitalio→**nextDigitalIO()****digitalio**→
nextDigitalIO()

YDigitalIO

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

`YDigitalIO * nextDigitalIO()`

Returns :

a pointer to a `YDigitalIO` object, corresponding to a digital IO port currently online, or a `null` pointer if there are no more digital IO ports to enumerate.

Triggers a pulse on a single bit for a specified duration.

```
int pulse( int bitno, int ms_duration)
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno the bit number; lowest bit has index 0

ms_duration desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**registerValueCallback()****digitalio**→
registerValueCallback()

YDigitalIO

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YDigitalIOValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

digitalio→**set_bitDirection()****YDigitalIO****digitalio**→**setBitDirection()****digitalio**→**set_bitDirection()**

Changes the direction of a single bit from the I/O port.

```
int set_bitDirection( int bitno, int bitdirection)
```

Parameters :

bitno the bit number; lowest bit has index 0

bitdirection direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_bitOpenDrain()****YDigitalIO****digitalio**→**setBitOpenDrain()****digitalio**→**set_bitOpenDrain()**

Changes the electrical interface of a single bit from the I/O port.

```
int set_bitOpenDrain( int bitno, int opendrain)
```

Parameters :

bitno the bit number; lowest bit has index 0

opendrain 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_bitPolarity()****YDigitalIO****digitalio**→**setBitPolarity()****digitalio**→**set_bitPolarity()**

Changes the polarity of a single bit from the I/O port.

```
int set_bitPolarity( int bitno, int bitpolarity)
```

Parameters :

bitno the bit number; lowest bit has index 0.

bitpolarity polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_bitState()****YDigitalIO****digitalio**→**setBitState()****digitalio**→
set_bitState()

Sets a single bit of the I/O port.

```
int set_bitState( int bitno, int bitstate)
```

Parameters :

bitno the bit number; lowest bit has index 0

bitstate the state of the bit (1 or 0)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_logicalName()****YDigitalIO****digitalio**→**setLogicalName()****digitalio**→**set_logicalName()**

Changes the logical name of the digital IO port.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the digital IO port.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

digitalio→set_outputVoltage()**YDigitalIO****digitalio→setOutputVoltage()**
digitalio→
set_outputVoltage()

Changes the voltage source used to drive output bits.

```
int set_outputVoltage( Y_OUTPUTVOLTAGE_enum newval)
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portDirection()****YDigitalIO****digitalio**→**setPortDirection()****digitalio**→**set_portDirection()**

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

```
int set_portDirection( int newval)
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portOpenDrain()****YDigitalIO****digitalio**→**setPortOpenDrain()****digitalio**→
set_portOpenDrain()

Changes the electrical interface for each bit of the port.

```
int set_portOpenDrain( int newval)
```

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the electrical interface for each bit of the port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→set_portPolarity()**YDigitalIO****digitalio→setPortPolarity()****digitalio→****set_portPolarity()**

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

```
int set_portPolarity( int newval)
```

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

Parameters :

newval an integer corresponding to the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portState()****YDigitalIO****digitalio**→**setPortState()****digitalio**→
set_portState()

Changes the digital IO port state: bit 0 represents input 0, and so on.

```
int set_portState( int newval)
```

This function has no effect on bits configured as input in `portDirection`.

Parameters :

newval an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_userdata()**

YDigitalIO

digitalio→**setUserData()****digitalio**→

set_userdata()

Stores a user context provided as argument in the `userData` attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

digitalio→**toggle_bitState()****digitalio**→
toggle_bitState()

YDigitalIO

Reverts a single bit of the I/O port.

```
int toggle_bitState( int bitno)
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.13. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_display.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDisplay = yoctolib.YDisplay;
php	require_once('yocto_display.php');
c++	#include "yocto_display.h"
m	#import "yocto_display.h"
pas	uses yocto_display;
vb	yocto_display.vb
cs	yocto_display.cs
java	import com.yoctopuce.YoctoAPI.YDisplay;
py	from yocto_display import *

Global functions

yFindDisplay(func)

Retrieves a display for a given identifier.

yFirstDisplay()

Starts the enumeration of displays currently accessible.

YDisplay methods

display→copyLayerContent(srcLayerId, dstLayerId)

Copies the whole content of a layer to another layer.

display→describe()

Returns a short text that describes unambiguously the instance of the display in the form
TYPE (NAME) =SERIAL . FUNCTIONID.

display→fade(brightness, duration)

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

display→get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

display→get_brightness()

Returns the luminosity of the module informative leds (from 0 to 100).

display→get_displayHeight()

Returns the display height, in pixels.

display→get_displayLayer(layerId)

Returns a YDisplayLayer object that can be used to draw on the specified layer.

display→get_displayType()

Returns the display type: monochrome, gray levels or full color.

display→get_displayWidth()

Returns the display width, in pixels.

display→get_enabled()

Returns true if the screen is powered, false otherwise.

display→get_errorMessage()

Returns the error message of the latest error with the display.

display→get_errorType()

Returns the numerical error code of the latest error with the display.

display→get_friendlyName()

Returns a global identifier of the display in the format `MODULE_NAME . FUNCTION_NAME`.

display→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

display→get_functionId()

Returns the hardware identifier of the display, without reference to the module.

display→get_hardwareId()

Returns the unique hardware identifier of the display in the form `SERIAL . FUNCTIONID`.

display→get_layerCount()

Returns the number of available layers to draw on.

display→get_layerHeight()

Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()

Returns the width of the layers to draw on, in pixels.

display→get_logicalName()

Returns the logical name of the display.

display→get_module()

Gets the `YModule` object for the device on which the function is located.

display→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

display→get_orientation()

Returns the currently selected display orientation.

display→get_startupSeq()

Returns the name of the sequence to play when the displayed is powered on.

display→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

display→isOnline()

Checks if the display is currently reachable, without raising any error.

display→isOnline_async(callback, context)

Checks if the display is currently reachable, without raising any error (asynchronous version).

display→load(msValidity)

Preloads the display cache with a specified validity duration.

display→load_async(msValidity, callback, context)

Preloads the display cache with a specified validity duration (asynchronous version).

display→newSequence()

Starts to record all display commands into a sequence, for later replay.

display→nextDisplay()

Continues the enumeration of displays started using `yFirstDisplay()`.

display→pauseSequence(delay_ms)

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

display→playSequence(sequenceName)

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

display→registerValueCallback(callback)

3. Reference

Registers the callback function that is invoked on every change of advertised value.

display→resetAll()

Clears the display screen and resets all display layers to their default state.

display→saveSequence(sequenceName)

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

display→set_brightness(newval)

Changes the brightness of the display.

display→set_enabled(newval)

Changes the power state of the display.

display→set_logicalName(newval)

Changes the logical name of the display.

display→set_orientation(newval)

Changes the display orientation.

display→set_startupSeq(newval)

Changes the name of the sequence to play when the displayed is powered on.

display→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

display→stopSequence()

Stops immediately any ongoing sequence replay.

display→swapLayerContent(layerIdA, layerIdB)

Swaps the whole content of two layers.

display→upload(pathname, content)

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

display→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDisplay.FindDisplay() yFindDisplay()yFindDisplay()

YDisplay

Retrieves a display for a given identifier.

```
YDisplay* yFindDisplay( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the display

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FirstDisplay()

YDisplay

yFirstDisplay()**yFirstDisplay()**

Starts the enumeration of displays currently accessible.

YDisplay* **yFirstDisplay()**

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

Returns :

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

display→copyLayerContent()**display→**
copyLayerContent ()

YDisplay

Copies the whole content of a layer to another layer.

```
int copyLayerContent( int srcLayerId, int dstLayerId)
```

The color and transparency of all the pixels from the destination layer are set to match the source pixels. This method only affects the displayed content, but does not change any property of the layer object. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

srcLayerId the identifier of the source layer (a number in range 0..layerCount-1)

dstLayerId the identifier of the destination layer (a number in range 0..layerCount-1)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→describe()**display→describe()****YDisplay**

Returns a short text that describes unambiguously the instance of the display in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the display (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

display→fade()**display→fade()****YDisplay**

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

```
int fade( int brightness, int duration)
```

Parameters :

brightness the new screen brightness

duration duration of the brightness transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→get_advertisedValue()

YDisplay

display→advertisedValue()**display→**

get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the display (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

display→get_brightness()**YDisplay****display→brightness()****display→get_brightness()**

Returns the luminosity of the module informative leds (from 0 to 100).

```
int get_brightness( )
```

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y_BRIGHTNESS_INVALID.

display→get_displayHeight()

YDisplay

display→displayHeight()**display→**

get_displayHeight()

Returns the display height, in pixels.

int **get_displayHeight()**

Returns :

an integer corresponding to the display height, in pixels

On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

display→**get_displayLayer()**
display→**displayLayer()****display**→
get_displayLayer()

YDisplay

Returns a YDisplayLayer object that can be used to draw on the specified layer.

YDisplayLayer* **get_displayLayer**(unsigned **layerId**)

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

Parameters :

layerId the identifier of the layer (a number in range 0..layerCount-1)

Returns :

an YDisplayLayer object

On failure, throws an exception or returns null.

display→get_displayType()

YDisplay

display→displayType()**display→**

get_displayType()

Returns the display type: monochrome, gray levels or full color.

Y_DISPLAYTYPE_enum **get_displayType()**

Returns :

a value among Y_DISPLAYTYPE_MONO, Y_DISPLAYTYPE_GRAY and Y_DISPLAYTYPE_RGB corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns Y_DISPLAYTYPE_INVALID.

display→**get_displayWidth()**
display→**displayWidth()****display**→
get_displayWidth()

YDisplay

Returns the display width, in pixels.

int **get_displayWidth()**

Returns :

an integer corresponding to the display width, in pixels

On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

display→get_enabled()

YDisplay

display→enabled()**display→get_enabled()**

Returns true if the screen is powered, false otherwise.

`Y_ENABLED_enum` **get_enabled()**

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

display→**get_errorMessage()****YDisplay****display**→**errorMessage()****display**→
get_errorMessage()

Returns the error message of the latest error with the display.

`string` **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the display object

display→get_errorType()

YDisplay

display→errorType()**display→get_errorType()**

Returns the numerical error code of the latest error with the display.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the display object

display→**get_friendlyName()**
display→**friendlyName()****display**→
get_friendlyName()

YDisplay

Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the display if they are defined, otherwise the serial number of the module and the hardware identifier of the display (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the display using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

display→**get_functionDescriptor()**

YDisplay

display→**functionDescriptor()****display**→

get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

display→get_functionId()**YDisplay****display→functionId()****display→get_functionId()**

Returns the hardware identifier of the display, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the display (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

display→get_hardwareId()

YDisplay

display→hardwareId()**display→get_hardwareId()**

Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the display (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

display→get_layerCount()**YDisplay****display→layerCount()****display→get_layerCount()**

Returns the number of available layers to draw on.

```
int get_layerCount( )
```

Returns :

an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns Y_LAYERCOUNT_INVALID.

display→get_layerHeight()

YDisplay

display→layerHeight()**display→**

get_layerHeight()

Returns the height of the layers to draw on, in pixels.

int get_layerHeight()

Returns :

an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

display→get_layerWidth()**YDisplay****display→layerWidth()****display→get_layerWidth()**

Returns the width of the layers to draw on, in pixels.

int **get_layerWidth()**

Returns :

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERWIDTH_INVALID.

display→get_logicalName()

YDisplay

display→logicalName()
display→get_logicalName()

Returns the logical name of the display.

string get_logicalName()

Returns :

a string corresponding to the logical name of the display. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

display→get_module()**YDisplay****display→module()****display→get_module()**

Gets the YModule object for the device on which the function is located.

```
YModule * get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

display→get_orientation()

YDisplay

display→orientation()**display→**
get_orientation()

Returns the currently selected display orientation.

`Y_ORIENTATION_enum` **get_orientation()**

Returns :

a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the currently selected display orientation

On failure, throws an exception or returns `Y_ORIENTATION_INVALID`.

display→get_startupSeq()**YDisplay****display→startupSeq()****display→get_startupSeq()**

Returns the name of the sequence to play when the displayed is powered on.

string **get_startupSeq()**

Returns :

a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns Y_STARTUPSEQ_INVALID.

display→get_userdata()

YDisplay

display→userdata()display→get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata()
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

display→isOnline()**display→isOnline()****YDisplay**

Checks if the display is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

Returns :

`true` if the display can be reached, and `false` otherwise

display→load()**display→load()****YDisplay**

Preloads the display cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

display→newSequence()**display→newSequence()**

YDisplay

Starts to record all display commands into a sequence, for later replay.

int newSequence()

The name used to store the sequence is specified when calling `saveSequence()`, once the recording is complete.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→nextDisplay()**display→nextDisplay()**

YDisplay

Continues the enumeration of displays started using `yFirstDisplay()`.

`YDisplay * nextDisplay()`

Returns :

a pointer to a `YDisplay` object, corresponding to a display currently online, or a `null` pointer if there are no more displays to enumerate.

display→pauseSequence()**display→**
pauseSequence ()**YDisplay**

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

```
int pauseSequence( int delay_ms)
```

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

Parameters :

delay_ms the duration to wait, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→playSequence()**display→**
playSequence()

YDisplay

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

```
int playSequence( string sequenceName)
```

Parameters :

sequenceName the name of the newly created sequence

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→registerValueCallback()**display→**
registerValueCallback()

YDisplay

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YDisplayValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

display→resetAll()**display→resetAll()**

YDisplay

Clears the display screen and resets all display layers to their default state.

`int resetAll()`

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**saveSequence()****display**→
saveSequence()

YDisplay

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```
int saveSequence( string sequenceName)
```

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_brightness()****YDisplay****display**→**setBrightness()****display**→
set_brightness()

Changes the brightness of the display.

```
int set_brightness( int newval)
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the brightness of the display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_enabled()****YDisplay****display**→**setEnabled()****display**→**set_enabled()**

Changes the power state of the display.

```
int set_enabled( Y_ENABLED_enum newval)
```

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the power state of the display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_logicalName()****YDisplay****display**→**setLogicalName()****display**→
set_logicalName()

Changes the logical name of the display.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the display.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

display→**set_orientation()**
display→**setOrientation()****display**→
set_orientation()

YDisplay

Changes the display orientation.

```
int set_orientation( Y_ORIENTATION_enum newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among Y_ORIENTATION_LEFT, Y_ORIENTATION_UP, Y_ORIENTATION_RIGHT and Y_ORIENTATION_DOWN corresponding to the display orientation

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_startupSeq()****YDisplay****display**→**setStartupSeq()****display**→**set_startupSeq()**

Changes the name of the sequence to play when the displayed is powered on.

```
int set_startupSeq( const string& newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the name of the sequence to play when the displayed is powered on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_userdata()**YDisplay****display→setUserData()****display→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

display→stopSequence()display→
stopSequence()

YDisplay

Stops immediately any ongoing sequence replay.

int stopSequence()

The display is left as is.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→swapLayerContent()**display→**
swapLayerContent ()

YDisplay

Swaps the whole content of two layers.

```
int swapLayerContent( int layerIdA, int layerIdB)
```

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between one hidden layer and a visible layer, this method makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

layerIdA the first layer (a number in range 0..layerCount-1)

layerIdB the second layer (a number in range 0..layerCount-1)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→upload()display→upload()**YDisplay**

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

```
int upload( string pathname, string content)
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.14. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_display.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDisplay = yoctolib.YDisplay;
php	require_once('yocto_display.php');
c++	#include "yocto_display.h"
m	#import "yocto_display.h"
pas	uses yocto_display;
vb	yocto_display.vb
cs	yocto_display.cs
java	import com.yoctopuce.YoctoAPI.YDisplay;
py	from yocto_display import *

YDisplayLayer methods

displaylayer→clear()

Erases the whole content of the layer (makes it fully transparent).

displaylayer→clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

displaylayer→consoleOut(text)

Outputs a message in the console area, and advances the console pointer accordingly.

displaylayer→drawBar(x1, y1, x2, y2)

Draws a filled rectangular bar at a specified position.

displaylayer→drawBitmap(x, y, w, bitmap, bgcolor)

Draws a bitmap at the specified position.

displaylayer→drawCircle(x, y, r)

Draws an empty circle at a specified position.

displaylayer→drawDisc(x, y, r)

Draws a filled disc at a given position.

displaylayer→drawImage(x, y, imagename)

Draws a GIF image at the specified position.

displaylayer→drawPixel(x, y)

Draws a single pixel at the specified position.

displaylayer→drawRect(x1, y1, x2, y2)

Draws an empty rectangle at a specified position.

displaylayer→drawText(x, y, anchor, text)

Draws a text string at the specified position.

displaylayer→get_display()

Gets parent YDisplay.

displaylayer→get_displayHeight()

Returns the display height, in pixels.

displaylayer→get_displayWidth()

Returns the display width, in pixels.

displaylayer→get_layerHeight()

Returns the height of the layers to draw on, in pixels.

displaylayer→get_layerWidth()

Returns the width of the layers to draw on, in pixels.

displaylayer→hide()

Hides the layer.

displaylayer→lineTo(x, y)

Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(x, y)

Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()

Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(color)

Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

displaylayer→selectFont(fontname)

Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(graylevel)

Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(mode)

Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(bgcol)

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(x1, y1, x2, y2)

Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(wordwrap)

Sets up the wrapping behaviour used by the `consoleOut` function.

displaylayer→setLayerPosition(x, y, scrollTime)

Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()

Shows the layer.

displaylayer→clear()**displaylayer→clear()****YDisplayLayer**

Erases the whole content of the layer (makes it fully transparent).

int clear()

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset()` instead.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→clearConsole()displaylayer→
clearConsole()

YDisplayLayer

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

int **clearConsole()**

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→consoleOut()**displaylayer→**
consoleOut ()**YDisplayLayer**

Outputs a message in the console area, and advances the console pointer accordingly.

```
int consoleOut( string text)
```

The console pointer position is automatically moved to the beginning of the next line when a newline character is met, or when the right margin is hit. When the new text to display extends below the lower margin, the console area is automatically scrolled up.

Parameters :

text the message to display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBar()**displaylayer→drawBar ()****YDisplayLayer**

Draws a filled rectangular bar at a specified position.

```
int drawBar( int x1, int y1, int x2, int y2)
```

Parameters :

- x1** the distance from left of layer to the left border of the rectangle, in pixels
- y1** the distance from top of layer to the top border of the rectangle, in pixels
- x2** the distance from left of layer to the right border of the rectangle, in pixels
- y2** the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawBitmap()displaylayer→
drawBitmap()**

YDisplayLayer

Draws a bitmap at the specified position.

```
int drawBitmap( int x, int y, int w, string bitmap, int bgcol)
```

The bitmap is provided as a binary object, where each pixel maps to a bit, from left to right and from top to bottom. The most significant bit of each byte maps to the leftmost pixel, and the least significant bit maps to the rightmost pixel. Bits set to 1 are drawn using the layer selected pen color. Bits set to 0 are drawn using the specified background gray level, unless -1 is specified, in which case they are not drawn at all (as if transparent).

Parameters :

- x** the distance from left of layer to the left of the bitmap, in pixels
- y** the distance from top of layer to the top of the bitmap, in pixels
- w** the width of the bitmap, in pixels
- bitmap** a binary object
- bgcol** the background gray level to use for zero bits (0 = black, 255 = white), or -1 to leave the pixels unchanged

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**drawCircle()****displaylayer**→
drawCircle()

YDisplayLayer

Draws an empty circle at a specified position.

```
int drawCircle( int x, int y, int r)
```

Parameters :

- x** the distance from left of layer to the center of the circle, in pixels
- y** the distance from top of layer to the center of the circle, in pixels
- r** the radius of the circle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawDisc()**displaylayer→**
drawDisc()

YDisplayLayer

Draws a filled disc at a given position.

```
int drawDisc( int x, int y, int r)
```

Parameters :

- x** the distance from left of layer to the center of the disc, in pixels
- y** the distance from top of layer to the center of the disc, in pixels
- r** the radius of the disc, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawImage()**displaylayer→**
drawImage ()

YDisplayLayer

Draws a GIF image at the specified position.

```
int drawImage( int x, int y, string imagename)
```

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

Parameters :

x the distance from left of layer to the left of the image, in pixels
y the distance from top of layer to the top of the image, in pixels
imagename the GIF file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**drawPixel()****displaylayer**→
drawPixel()

YDisplayLayer

Draws a single pixel at the specified position.

```
int drawPixel( int x, int y)
```

Parameters :

- x** the distance from left of layer, in pixels
- y** the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawRect()displaylayer→
drawRect ()

YDisplayLayer

Draws an empty rectangle at a specified position.

```
int drawRect( int x1, int y1, int x2, int y2)
```

Parameters :

- x1** the distance from left of layer to the left border of the rectangle, in pixels
- y1** the distance from top of layer to the top border of the rectangle, in pixels
- x2** the distance from left of layer to the right border of the rectangle, in pixels
- y2** the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawText()**displaylayer→drawText ()**

YDisplayLayer

Draws a text string at the specified position.

```
int drawText( int x, int y, Y_ALIGN anchor, string text)
```

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

Parameters :

- x** the distance from left of layer to the text anchor point, in pixels
- y** the distance from top of layer to the text anchor point, in pixels
- anchor** the text anchor point, chosen among the Y_ALIGN enumeration: Y_ALIGN_TOP_LEFT, Y_ALIGN_CENTER_LEFT, Y_ALIGN_BASELINE_LEFT, Y_ALIGN_BOTTOM_LEFT, Y_ALIGN_TOP_CENTER, Y_ALIGN_CENTER, Y_ALIGN_BASELINE_CENTER, Y_ALIGN_BOTTOM_CENTER, Y_ALIGN_TOP_DECIMAL, Y_ALIGN_CENTER_DECIMAL, Y_ALIGN_BASELINE_DECIMAL, Y_ALIGN_BOTTOM_DECIMAL, Y_ALIGN_TOP_RIGHT, Y_ALIGN_CENTER_RIGHT, Y_ALIGN_BASELINE_RIGHT, Y_ALIGN_BOTTOM_RIGHT.
- text** the text string to draw

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→get_display()

YDisplayLayer

**displaylayer→display()displaylayer→
get_display()**

Gets parent YDisplay.

YDisplay* **get_display()**

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :

an YDisplay object

displaylayer→**get_displayHeight()****YDisplayLayer****displaylayer**→**displayHeight()****displaylayer**→
get_displayHeight()

Returns the display height, in pixels.

int **get_displayHeight()**

Returns :

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth()

YDisplayLayer

displaylayer→displayWidth()displaylayer→

get_displayWidth()

Returns the display width, in pixels.

int get_displayWidth()

Returns :

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→**get_layerHeight()****YDisplayLayer****displaylayer**→**layerHeight()****displaylayer**→
get_layerHeight()

Returns the height of the layers to draw on, in pixels.

int **get_layerHeight()**

Returns :

an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth()

YDisplayLayer

**displaylayer→layerWidth()displaylayer→
get_layerWidth()**

Returns the width of the layers to draw on, in pixels.

int get_layerWidth()

Returns :

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERWIDTH_INVALID.

displaylayer→hide()displaylayer→hide()

YDisplayLayer

Hides the layer.

```
int hide( )
```

The state of the layer is perserved but the layer is not displayed on the screen until the next call to `unhide()`. Hiding the layer can positively affect the drawing speed, since it postpones the rendering until all operations are completed (double-buffering).

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→lineTo()**displaylayer→lineTo()****YDisplayLayer**

Draws a line from current drawing pointer position to the specified position.

```
int lineTo( int x, int y)
```

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

Parameters :

- x** the distance from left of layer to the end point of the line, in pixels
- y** the distance from top of layer to the end point of the line, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→moveTo()**displaylayer→moveTo ()****YDisplayLayer**

Moves the drawing pointer of this layer to the specified position.

```
int moveTo( int x, int y)
```

Parameters :

- x** the distance from left of layer, in pixels
- y** the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→reset()displaylayer→reset()**YDisplayLayer**

Reverts the layer to its initial state (fully transparent, default settings).

```
int reset( )
```

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectColorPen()**displaylayer→**
selectColorPen()

YDisplayLayer

Selects the pen color for all subsequent drawing functions, including text drawing.

```
int selectColorPen( int color)
```

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**selectEraser()****displaylayer**→
selectEraser()

YDisplayLayer

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

int **selectEraser()**

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectFont()**displaylayer→**
selectFont()**YDisplayLayer**

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```
int selectFont( string fontname)
```

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

Parameters :

fontname the font file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**selectGrayPen()****displaylayer**→
selectGrayPen()**YDisplayLayer**

Selects the pen gray level for all subsequent drawing functions, including text drawing.

```
int selectGrayPen( int graylevel)
```

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the highest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

Parameters :

graylevel the desired gray level, from 0 to 255

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**setAntialiasingMode()****displaylayer**→
setAntialiasingMode()

YDisplayLayer

Enables or disables anti-aliasing for drawing oblique lines and circles.

```
int setAntialiasingMode( bool mode)
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

Parameters :

mode true to enable antialiasing, false to disable it.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleBackground()

YDisplayLayer

displaylayer→setConsoleBackground()

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

```
int setConsoleBackground( int bgcol)
```

Parameters :

bgcol the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**setConsoleMargins()****displaylayer**→
setConsoleMargins()

YDisplayLayer

Sets up display margins for the `consoleOut` function.

```
int setConsoleMargins( int x1, int y1, int x2, int y2)
```

Parameters :

- x1** the distance from left of layer to the left margin, in pixels
- y1** the distance from top of layer to the top margin, in pixels
- x2** the distance from left of layer to the right margin, in pixels
- y2** the distance from top of layer to the bottom margin, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**setConsoleWordWrap()****displaylayer**
→**setConsoleWordWrap()****YDisplayLayer**

Sets up the wrapping behaviour used by the `consoleOut` function.

```
int setConsoleWordWrap( bool wordwrap )
```

Parameters :

wordwrap `true` to wrap only between words, `false` to wrap on the last column anyway.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**setLayerPosition()****displaylayer**→
setLayerPosition()

YDisplayLayer

Sets the position of the layer relative to the display upper left corner.

```
int setLayerPosition( int x, int y, int scrollTime)
```

When smooth scrolling is used, the display offset of the layer is automatically updated during the next milliseconds to animate the move of the layer.

Parameters :

- x** the distance from left of display to the upper left corner of the layer
- y** the distance from top of display to the upper left corner of the layer
- scrollTime** number of milliseconds to use for smooth scrolling, or 0 if the scrolling should be immediate.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→unhide()displaylayer→unhide()

YDisplayLayer

Shows the layer.

```
int unhide( )
```

Shows the layer again after a hide command.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.15. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_dualpower.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDualPower = yoctolib.YDualPower;
php	require_once('yocto_dualpower.php');
c++	#include "yocto_dualpower.h"
m	#import "yocto_dualpower.h"
pas	uses yocto_dualpower;
vb	yocto_dualpower.vb
cs	yocto_dualpower.cs
java	import com.yoctopuce.YoctoAPI.YDualPower;
py	from yocto_dualpower import *

Global functions

yFindDualPower(func)

Retrieves a dual power control for a given identifier.

yFirstDualPower()

Starts the enumeration of dual power controls currently accessible.

YDualPower methods

dualpower→describe()

Returns a short text that describes unambiguously the instance of the power control in the form TYPE (NAME) = SERIAL . FUNCTIONID.

dualpower→get_advertisedValue()

Returns the current value of the power control (no more than 6 characters).

dualpower→get_errorMessage()

Returns the error message of the latest error with the power control.

dualpower→get_errorType()

Returns the numerical error code of the latest error with the power control.

dualpower→get_extVoltage()

Returns the measured voltage on the external power source, in millivolts.

dualpower→get_friendlyName()

Returns a global identifier of the power control in the format MODULE_NAME . FUNCTION_NAME.

dualpower→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

dualpower→get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

dualpower→get_hardwareId()

Returns the unique hardware identifier of the power control in the form SERIAL . FUNCTIONID.

dualpower→get_logicalName()

Returns the logical name of the power control.

dualpower→get_module()

	Gets the <code>YModule</code> object for the device on which the function is located.
<code>dualpower→get_module_async(callback, context)</code>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<code>dualpower→get_powerControl()</code>	Returns the selected power source for module functions that require lots of current.
<code>dualpower→get_powerState()</code>	Returns the current power source for module functions that require lots of current.
<code>dualpower→get_userData()</code>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<code>dualpower→isOnline()</code>	Checks if the power control is currently reachable, without raising any error.
<code>dualpower→isOnline_async(callback, context)</code>	Checks if the power control is currently reachable, without raising any error (asynchronous version).
<code>dualpower→load(msValidity)</code>	Preloads the power control cache with a specified validity duration.
<code>dualpower→load_async(msValidity, callback, context)</code>	Preloads the power control cache with a specified validity duration (asynchronous version).
<code>dualpower→nextDualPower()</code>	Continues the enumeration of dual power controls started using <code>yFirstDualPower()</code> .
<code>dualpower→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>dualpower→set_logicalName(newval)</code>	Changes the logical name of the power control.
<code>dualpower→set_powerControl(newval)</code>	Changes the selected power source for module functions that require lots of current.
<code>dualpower→set_userData(data)</code>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<code>dualpower→wait_async(callback, context)</code>	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDualPower.FindDualPower()**YDualPower****yFindDualPower()****yFindDualPower()**

Retrieves a dual power control for a given identifier.

```
YDualPower* yFindDualPower( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDualPower.isOnline()` to test if the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

YDualPower.FirstDualPower()

YDualPower

yFirstDualPower()`yFirstDualPower()`

Starts the enumeration of dual power controls currently accessible.

`YDualPower* yFirstDualPower()`

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()

YDualPower

Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the power control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

dualpower→**get_advertisedValue()**

YDualPower

dualpower→**advertisedValue()****dualpower**→

get_advertisedValue()

Returns the current value of the power control (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the power control (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

dualpower→get_errorMessage()**YDualPower****dualpower→errorMessage()****dualpower→**
get_errorMessage()

Returns the error message of the latest error with the power control.

`string` **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the power control object

dualpower→**get_errorType()**

YDualPower

dualpower→**errorType()****dualpower**→

get_errorType()

Returns the numerical error code of the latest error with the power control.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the power control object

dualpower→get_extVoltage()**YDualPower****dualpower→extVoltage()****dualpower→****get_extVoltage()**

Returns the measured voltage on the external power source, in millivolts.

int **get_extVoltage()**

Returns :

an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns Y_EXTVOLTAGE_INVALID.

dualpower→**get_friendlyName()****YDualPower****dualpower**→**friendlyName()****dualpower**→**get_friendlyName()**

Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the power control if they are defined, otherwise the serial number of the module and the hardware identifier of the power control (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the power control using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

dualpower→**get_functionDescriptor()****YDualPower****dualpower**→**functionDescriptor()****dualpower**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

dualpower→**get_functionId()**

YDualPower

dualpower→**functionId()****dualpower**→
get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the power control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

dualpower→**get_hardwareId()****YDualPower****dualpower**→**hardwareId()****dualpower**→
get_hardwareId()

Returns the unique hardware identifier of the power control in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the power control. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the power control (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

dualpower→**get_logicalName()**

YDualPower

dualpower→**logicalName()****dualpower**→
get_logicalName()

Returns the logical name of the power control.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the power control. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

dualpower→get_module()**YDualPower****dualpower→module()****dualpower→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

dualpower→**get_powerControl()****YDualPower****dualpower**→**powerControl()****dualpower**→**get_powerControl()**

Returns the selected power source for module functions that require lots of current.

Y_POWERCONTROL_enum **get_powerControl()**

Returns :

a value among Y_POWERCONTROL_AUTO, Y_POWERCONTROL_FROM_USB, Y_POWERCONTROL_FROM_EXT and Y_POWERCONTROL_OFF corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERCONTROL_INVALID.

dualpower→get_powerState()**YDualPower****dualpower→powerState()****dualpower→**
get_powerState()

Returns the current power source for module functions that require lots of current.

[Y_POWERSTATE_enum](#) **get_powerState()**

Returns :

a value among Y_POWERSTATE_OFF, Y_POWERSTATE_FROM_USB and Y_POWERSTATE_FROM_EXT corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERSTATE_INVALID.

dualpower→get_userdata()

YDualPower

dualpower→userdata()**dualpower→**

get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

dualpower→isOnline()**dualpower→isOnline()****YDualPower**

Checks if the power control is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power control.

Returns :

`true` if the power control can be reached, and `false` otherwise

dualpower→load()**dualpower→load()****YDualPower**

Preloads the power control cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

dualpower→**nextDualPower()****dualpower**→
nextDualPower ()

YDualPower

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()**dualpower→**
registerValueCallback()

YDualPower

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YDualPowerValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

dualpower→set_logicalName()**YDualPower****dualpower→setLogicalName()**
dualpower→set_logicalName()

Changes the logical name of the power control.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the power control.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

dualpower→**set_powerControl()****YDualPower****dualpower**→**setPowerControl()****dualpower**→
set_powerControl()

Changes the selected power source for module functions that require lots of current.

```
int set_powerControl( Y_POWERCONTROL_enum newval)
```

Parameters :

newval a value among Y_POWERCONTROL_AUTO, Y_POWERCONTROL_FROM_USB, Y_POWERCONTROL_FROM_EXT and Y_POWERCONTROL_OFF corresponding to the selected power source for module functions that require lots of current

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

dualpower→set_userdata()**YDualPower****dualpower→setUserData()**
dualpower→
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.16. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_files.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YFiles = yoctolib.YFiles;
php	require_once('yocto_files.php');
c++	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *

Global functions

yFindFiles(func)

Retrieves a filesystem for a given identifier.

yFirstFiles()

Starts the enumeration of filesystems currently accessible.

YFiles methods

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form TYPE (NAME) =SERIAL . FUNCTIONID.

files→download(pathname)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(pathname, callback, context)

Downloads the requested file and returns a binary buffer with its content.

files→format_fs()

Reinitializes the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

Returns a global identifier of the filesystem in the format MODULE_NAME . FUNCTION_NAME.

files→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

files→get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

files→get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

files→get_list(pattern)

Returns a list of `YFileRecord` objects that describe files currently loaded in the filesystem.

files→get_logicalName()

Returns the logical name of the filesystem.

files→get_module()

Gets the `YModule` object for the device on which the function is located.

files→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

files→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

files→isOnline()

Checks if the filesystem is currently reachable, without raising any error.

files→isOnline_async(callback, context)

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

files→load(msValidity)

Preloads the filesystem cache with a specified validity duration.

files→load_async(msValidity, callback, context)

Preloads the filesystem cache with a specified validity duration (asynchronous version).

files→nextFiles()

Continues the enumeration of filesystems started using `yFirstFiles()`.

files→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

files→remove(pathname)

Deletes a file, given by its full path name, from the filesystem.

files→set_logicalName(newval)

Changes the logical name of the filesystem.

files→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

files→upload(pathname, content)

Uploads a file to the filesystem, to the specified full path name.

files→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YFiles.FindFiles()**YFiles****yFindFiles()****yFindFiles()**

Retrieves a filesystem for a given identifier.

```
YFiles* yFindFiles( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the filesystem

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles()**YFiles****yFirstFiles()****yFirstFiles()**

Starts the enumeration of filesystems currently accessible.

YFiles* **yFirstFiles()**

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

files→describe()**files→describe()****YFiles**

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the filesystem (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

files→download()**files→download()****YFiles**

Downloads the requested file and returns a binary buffer with its content.

```
string download( string pathname)
```

Parameters :

pathname path and name of the file to download

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

files→**format_fs()****files**→**format_fs()****YFiles**

Reinitializes the filesystem to its clean, unfragmented, empty state.

```
int format_fs( )
```

All files previously uploaded are permanently lost.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→**get_advertisedValue()**
files→**advertisedValue()****files**→
get_advertisedValue()

YFiles

Returns the current value of the filesystem (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the filesystem (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

files→get_errorMessage()

YFiles

files→errorMessage()**files→get_errorMessage()**

Returns the error message of the latest error with the filesystem.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the filesystem object

files→**get_errorType()****YFiles****files**→**errorType()****files**→**get_errorType()**

Returns the numerical error code of the latest error with the filesystem.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the filesystem object

files→get_filesCount()

YFiles

files→filesCount()**files→get_filesCount()**

Returns the number of files currently loaded in the filesystem.

int get_filesCount()

Returns :

an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns Y_FILESCOUNT_INVALID.

files→get_freeSpace()**YFiles****files→freeSpace()****files→get_freeSpace()**

Returns the free space for uploading new files to the filesystem, in bytes.

int **get_freeSpace()**

Returns :

an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns Y_FREESPACE_INVALID.

files→get_friendlyName()

YFiles

files→friendlyName()**files→get_friendlyName()**

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

files→**get_functionDescriptor()**
files→**functionDescriptor()****files**→
get_functionDescriptor()

YFiles

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

files→get_functionId()

YFiles

files→functionId()**files→get_functionId()**

Returns the hardware identifier of the filesystem, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the filesystem (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

files→**get_hardwareId()****YFiles****files**→**hardwareId()****files**→**get_hardwareId()**

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the filesystem (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

files→get_list()**files→list()****files→get_list()**

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
vector<YFileRecord> get_list( string pattern)
```

Parameters :

pattern an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

Returns :

a list of YFileRecord objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName()**YFiles****files→logicalName()****files→get_logicalName()**

Returns the logical name of the filesystem.

```
string get_logicalName( )
```

Returns :

a string corresponding to the logical name of the filesystem. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

files→get_module()

YFiles

files→module()**files→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

files→get_userdata()**YFiles****files→userdata()****files→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

files→isOnline()**files→isOnline()****YFiles**

Checks if the filesystem is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

Returns :

`true` if the filesystem can be reached, and `false` otherwise

files→load()**files→load()****YFiles**

Preloads the filesystem cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

files→**nextFiles()****files**→**nextFiles()**

YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

YFiles * **nextFiles()**

Returns :

a pointer to a YFiles object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

files→registerValueCallback()**files→**
registerValueCallback()

YFiles

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YFilesValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

files→remove()**files→remove()****YFiles**

Deletes a file, given by its full path name, from the filesystem.

```
int remove( string pathname)
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→**set_logicalName()****YFiles****files**→**setLogicalName()****files**→**set_logicalName()**

Changes the logical name of the filesystem.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the filesystem.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

files→set_userdata()

YFiles

files→setUserData()**files→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

files→upload()files→upload()

YFiles

Uploads a file to the filesystem, to the specified full path name.

```
int upload( string pathname, string content)
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.17. GenericSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_genericsensor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YGenericSensor = yoctolib.YGenericSensor;
php	require_once('yocto_genericsensor.php');
c++	#include "yocto_genericsensor.h"
m	#import "yocto_genericsensor.h"
pas	uses yocto_genericsensor;
vb	yocto_genericsensor.vb
cs	yocto_genericsensor.cs
java	import com.yoctopuce.YoctoAPI.YGenericSensor;
py	from yocto_genericsensor import *

Global functions

yFindGenericSensor(func)

Retrieves a generic sensor for a given identifier.

yFirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

YGenericSensor methods

genericsensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

genericsensor→describe()

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

genericsensor→get_advertisedValue()

Returns the current value of the generic sensor (no more than 6 characters).

genericsensor→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

genericsensor→get_currentValue()

Returns the current measured value.

genericsensor→get_errorMessage()

Returns the error message of the latest error with the generic sensor.

genericsensor→get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

genericsensor→get_friendlyName()

Returns a global identifier of the generic sensor in the format `MODULE_NAME . FUNCTION_NAME`.

genericsensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

genericsensor→get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

genericsensor→get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form `SERIAL . FUNCTIONID`.

genericsensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

genericsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

genericsensor→get_logicalName()

Returns the logical name of the generic sensor.

genericsensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

genericsensor→get_module()

Gets the YModule object for the device on which the function is located.

genericsensor→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

genericsensor→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

genericsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

genericsensor→get_resolution()

Returns the resolution of the measured values.

genericsensor→get_signalRange()

Returns the electric signal range used by the sensor.

genericsensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericsensor→get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

genericsensor→get_unit()

Returns the measuring unit for the measure.

genericsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

genericsensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericsensor→isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

genericsensor→isOnline_async(callback, context)

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

genericsensor→load(msValidity)

Preloads the generic sensor cache with a specified validity duration.

genericsensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

genericsensor→load_async(msValidity, callback, context)

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

genericsensor→nextGenericSensor()

Continues the enumeration of generic sensors started using yFirstGenericSensor().

genericsensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

genericsensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

genericsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

genericsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

genericsensor→set_logicalName(newval)

Changes the logical name of the generic sensor.

genericsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

genericsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

genericsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

genericsensor→set_signalRange(newval)

Changes the electric signal range used by the sensor.

genericsensor→set_unit(newval)

Changes the measuring unit for the measured value.

genericsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

genericsensor→set_valueRange(newval)

Changes the physical value range measured by the sensor.

genericsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGenericSensor.FindGenericSensor() **yFindGenericSensor()****yFindGenericSensor()**

YGenericSensor

Retrieves a generic sensor for a given identifier.

```
YGenericSensor* yFindGenericSensor( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the generic sensor

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FirstGenericSensor()

YGenericSensor

yFirstGenericSensor()`yFirstGenericSensor()`

Starts the enumeration of generic sensors currently accessible.

`YGenericSensor*` **yFirstGenericSensor()**

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

Returns :

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a `null` pointer if there are none.

genericsensor→calibrateFromPoints()**YGenericSensor****genericsensor→calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→describe()genericsensor→
describe()

YGenericSensor

Returns a short text that describes unambiguously the instance of the generic sensor in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the generic sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

genericsensor→get_advertisedValue()**YGenericSensor****genericsensor→advertisedValue()****genericsensor→**
get_advertisedValue()

Returns the current value of the generic sensor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the generic sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

genericsensor→**get_currentRawValue()**

YGenericSensor

genericsensor→**currentRawValue()****genericsensor**

→**get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

genericsensor→get_currentValue()**YGenericSensor****genericsensor→currentValue()****genericsensor→**
get_currentValue()

Returns the current measured value.

`double` **get_currentValue()**

Returns :

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

genericsensor→get_errorMessage()

YGenericSensor

genericsensor→errorMessage()**genericsensor→**
get_errorMessage()

Returns the error message of the latest error with the generic sensor.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the generic sensor object

genericsensor→get_errorType()**YGenericSensor****genericsensor→errorType()****genericsensor→**
get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the generic sensor object

genericsensor→get_friendlyName()

YGenericSensor

genericsensor→friendlyName()**genericsensor→**
get_friendlyName()

Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the generic sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the generic sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the generic sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

genericsensor→**get_functionDescriptor()****YGenericSensor****genericsensor**→**functionDescriptor()****genericsensor**→**get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

genericsensor→get_functionId()

YGenericSensor

genericsensor→functionId()**genericsensor→**
get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

string get_functionId()

For example `relay1`

Returns :

a string that identifies the generic sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

genericsensor→**get_hardwareId()****YGenericSensor****genericsensor**→**hardwareId()****genericsensor**→
get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the generic sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the generic sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

genericsensor→get_highestValue()

YGenericSensor

genericsensor→highestValue()**genericsensor→**
get_highestValue()

Returns the maximal value observed for the measure since the device was started.

`double get_highestValue()`

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

genericsensor→get_logFrequency()**YGenericSensor****genericsensor→logFrequency()****genericsensor→**
get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

genericsensor→**get_logicalName()**

YGenericSensor

genericsensor→**logicalName()****genericsensor**→
get_logicalName()

Returns the logical name of the generic sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the generic sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

genericSensor→get_lowestValue()**YGenericSensor****genericSensor→lowestValue()**
genericSensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

genericsensor→get_module()

YGenericSensor

genericsensor→module()**genericsensor→**
get_module()

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

genericsensor→**get_recordedData()****YGenericSensor****genericsensor**→**recordedData()****genericsensor**→
get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

genericsensor→get_reportFrequency()

YGenericSensor

genericsensor→reportFrequency()genericsensor→

get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

genericsensor→get_resolution()**YGenericSensor****genericsensor→resolution()****genericsensor→**
get_resolution()

Returns the resolution of the measured values.

`double` **get_resolution()**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

genericsensor→get_signalRange()

YGenericSensor

genericsensor→signalRange()**genericsensor→**
get_signalRange()

Returns the electric signal range used by the sensor.

string get_signalRange()

Returns :

a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns Y_SIGNALRANGE_INVALID.

genericsensor→**get_signalUnit()****YGenericSensor****genericsensor**→**signalUnit()****genericsensor**→
get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

string **get_signalUnit()**

Returns :

a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALUNIT_INVALID.

genericsensor→get_signalValue()

YGenericSensor

genericsensor→signalValue()**genericsensor→**
get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

double get_signalValue()

Returns :

a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALVALUE_INVALID.

genericsensor→get_unit()**YGenericSensor****genericsensor→unit()****genericsensor→get_unit()**

Returns the measuring unit for the measure.

string **get_unit**()

Returns :

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns Y_UNIT_INVALID.

genericsensor→get_userdata()

YGenericSensor

genericsensor→userdata()**genericsensor→**
get_userdata()

Returns the value of the `userData` attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

genericsensor→get_valueRange()**YGenericSensor****genericsensor→valueRange()****genericsensor→**
get_valueRange()

Returns the physical value range measured by the sensor.

string **get_valueRange()**

Returns :

a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns Y_VALUERANGE_INVALID.

genericSensor→**isOnline()****genericSensor**→
isOnline()

YGenericSensor

Checks if the generic sensor is currently reachable, without raising any error.

bool **isOnline()**

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the generic sensor.

Returns :

`true` if the generic sensor can be reached, and `false` otherwise

generic_sensor→load()**generic_sensor→load()****YGenericSensor**

Preloads the generic sensor cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

genericsensor→loadCalibrationPoints()**YGenericSensor****genericsensor→loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericSensor→nextGenericSensor()**YGenericSensor****genericSensor→nextGenericSensor()**

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

`YGenericSensor * nextGenericSensor()`

Returns :

a pointer to a `YGenericSensor` object, corresponding to a generic sensor currently online, or a `null` pointer if there are no more generic sensors to enumerate.

genericsensor→registerTimedReportCallback()**YGenericSensor****genericsensor→****registerTimedReportCallback()**

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YGenericSensorTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

genericsensor→registerValueCallback()**YGenericSensor****genericsensor→registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YGenericSensorValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

genericsensor→**set_highestValue()**

YGenericSensor

genericsensor→**setHighestValue()****genericsensor**→
set_highestValue()

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→**set_logFrequency()****YGenericSensor****genericsensor**→**setLogFrequency()****genericsensor**
→**set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→**set_logicalName()****YGenericSensor****genericsensor**→**setLogicalName()****genericsensor**→
set_logicalName()

Changes the logical name of the generic sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the generic sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

genericsensor→set_lowestValue()**YGenericSensor****genericsensor→setLowestValue()****genericsensor→**
set_lowestValue()

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_reportFrequency()**YGenericSensor****genericsensor→setReportFrequency()****genericsensor→set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_resolution()**YGenericSensor****genericsensor→setResolution()****genericsensor→set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→**set_signalRange()****YGenericSensor****genericsensor**→**setSignalRange()****genericsensor**→**set_signalRange()**

Changes the electric signal range used by the sensor.

```
int set_signalRange( const string& newval)
```

Parameters :

newval a string corresponding to the electric signal range used by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→**set_unit()****YGenericSensor****genericsensor**→**setUnit()****genericsensor**→
set_unit()

Changes the measuring unit for the measured value.

```
int set_unit( const string& newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the measuring unit for the measured value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_userdata()

YGenericSensor

genericsensor→setUserData()**genericsensor→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

genericsensor→set_valueRange()**YGenericSensor****genericsensor→setValueRange()****genericsensor→set_valueRange()**

Changes the physical value range measured by the sensor.

```
int set_valueRange( const string& newval)
```

The range change may have a side effect on the display resolution, as it may be adapted automatically.

Parameters :

newval a string corresponding to the physical value range measured by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.18. Gyroscope function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YGyro = yoctolib.YGyro;
php	require_once('yocto_gyro.php');
c++	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

Global functions

yFindGyro(func)

Retrieves a gyroscope for a given identifier.

yFirstGyro()

Starts the enumeration of gyroscopes currently accessible.

YGyro methods

gyro→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

gyro→describe()

Returns a short text that describes unambiguously the instance of the gyroscope in the form TYPE (NAME) = SERIAL . FUNCTIONID.

gyro→get_advertisedValue()

Returns the current value of the gyroscope (no more than 6 characters).

gyro→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

gyro→get_currentValue()

Returns the current value of the angular velocity.

gyro→get_errorMessage()

Returns the error message of the latest error with the gyroscope.

gyro→get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

gyro→get_friendlyName()

Returns a global identifier of the gyroscope in the format MODULE_NAME . FUNCTION_NAME.

gyro→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

gyro→get_functionId()

Returns the hardware identifier of the gyroscope, without reference to the module.

gyro→get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form SERIAL . FUNCTIONID.

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

gyro→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

gyro→get_logicalName()

Returns the logical name of the gyroscope.

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

gyro→get_module()

Gets the YModule object for the device on which the function is located.

gyro→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionW()

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionX()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionY()

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionZ()

Returns the z component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

gyro→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

gyro→get_resolution()

Returns the resolution of the measured values.

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_unit()

Returns the measuring unit for the angular velocity.

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

gyro→get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

gyro→get_yValue()

3. Reference

	Returns the angular velocity around the Y axis of the device, as a floating point number.
gyro→get_zValue()	Returns the angular velocity around the Z axis of the device, as a floating point number.
gyro→isOnline()	Checks if the gyroscope is currently reachable, without raising any error.
gyro→isOnline_async(callback, context)	Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).
gyro→load(msValidity)	Preloads the gyroscope cache with a specified validity duration.
gyro→loadCalibrationPoints(rawValues, refValues)	Retrieves error correction data points previously entered using the method <code>calibrateFromPoints</code> .
gyro→load_async(msValidity, callback, context)	Preloads the gyroscope cache with a specified validity duration (asynchronous version).
gyro→nextGyro()	Continues the enumeration of gyroscopes started using <code>yFirstGyro()</code> .
gyro→registerAnglesCallback(callback)	Registers a callback function that will be invoked each time that the estimated device orientation has changed.
gyro→registerQuaternionCallback(callback)	Registers a callback function that will be invoked each time that the estimated device orientation has changed.
gyro→registerTimedReportCallback(callback)	Registers the callback function that is invoked on every periodic timed notification.
gyro→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
gyro→set_highestValue(newval)	Changes the recorded maximal value observed.
gyro→set_logFrequency(newval)	Changes the datalogger recording frequency for this function.
gyro→set_logicalName(newval)	Changes the logical name of the gyroscope.
gyro→set_lowestValue(newval)	Changes the recorded minimal value observed.
gyro→set_reportFrequency(newval)	Changes the timed value notification frequency for this function.
gyro→set_resolution(newval)	Changes the resolution of the measured physical values.
gyro→set_userData(data)	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
gyro→wait_async(callback, context)	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGyro.FindGyro() yFindGyro()yFindGyro()

YGyro

Retrieves a gyroscope for a given identifier.

```
YGyro* yFindGyro( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the gyroscope is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGyro.isOnline()` to test if the gyroscope is indeed online at a given time. In case of ambiguity when looking for a gyroscope by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the gyroscope

Returns :

a `YGyro` object allowing you to drive the gyroscope.

YGyro.FirstGyro()

YGyro

yFirstGyro()**yFirstGyro()**

Starts the enumeration of gyroscopes currently accessible.

YGyro* yFirstGyro()

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

Returns :

a pointer to a `YGyro` object, corresponding to the first gyro currently online, or a `null` pointer if there are none.

gyro→calibrateFromPoints()**gyro→**
calibrateFromPoints()

YGyro

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→describe()**gyro→describe()****YGyro**

Returns a short text that describes unambiguously the instance of the gyroscope in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the gyroscope (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

gyro→**get_advertisedValue()**
gyro→**advertisedValue()****gyro**→
get_advertisedValue()

YGyro

Returns the current value of the gyroscope (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the gyroscope (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

gyro→get_currentRawValue()

YGyro

gyro→currentRawValue()**gyro→**

get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double get_currentRawValue()

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

gyro→get_currentValue()**YGyro****gyro→currentValue()****gyro→get_currentValue()**

Returns the current value of the angular velocity.

double **get_currentValue()**

Returns :

a floating point number corresponding to the current value of the angular velocity

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

gyro→get_errorMessage()

YGyro

gyro→errorMessage()**gyro→get_errorMessage()**

Returns the error message of the latest error with the gyroscope.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the gyroscope object

gyro→get_errorType()**YGyro****gyro→errorType()****gyro→get_errorType()**

Returns the numerical error code of the latest error with the gyroscope.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the gyroscope object

gyro→get_friendlyName()

YGyro

gyro→friendlyName()**gyro→get_friendlyName()**

Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the gyroscope if they are defined, otherwise the serial number of the module and the hardware identifier of the gyroscope (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the gyroscope using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

gyro→get_functionDescriptor()
gyro→functionDescriptor()**gyro→**
get_functionDescriptor()

YGyro

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

gyro→get_functionId()

YGyro

gyro→functionId()**gyro→get_functionId()**

Returns the hardware identifier of the gyroscope, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the gyroscope (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

gyro→get_hardwareId()**YGyro****gyro→hardwareId()****gyro→get_hardwareId()**

Returns the unique hardware identifier of the gyroscope in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the gyroscope. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the gyroscope (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

gyro→get_heading()

YGyro

gyro→heading()**gyro→get_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_heading()**

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to heading in degrees, between 0 and 360.

gyro→get_highestValue()**YGyro****gyro→highestValue()****gyro→get_highestValue()**

Returns the maximal value observed for the angular velocity since the device was started.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

gyro→get_logFrequency()

YGyro

gyro→logFrequency()**gyro→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

gyro→get_logicalName()**YGyro****gyro→logicalName()****gyro→get_logicalName()**

Returns the logical name of the gyroscope.

```
string get_logicalName( )
```

Returns :

a string corresponding to the logical name of the gyroscope. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

gyro→get_lowestValue()

YGyro

gyro→lowestValue()**gyro→get_lowestValue()**

Returns the minimal value observed for the angular velocity since the device was started.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

gyro→get_module()**YGyro****gyro→module()****gyro→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

gyro→get_pitch()

YGyro

gyro→pitch()**gyro→get_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_pitch()** ()

The axis corresponding to the pitch angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

gyro→get_quaternionW()**YGyro****gyro→quaternionW()****gyro→get_quaternionW()**

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
double get_quaternionW( )
```

Returns :

a floating-point number corresponding to the w component of the quaternion.

gyro→get_quaternionX()

YGyro

gyro→quaternionX()**gyro→get_quaternionX()**

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

`double get_quaternionX()`

The x component is mostly correlated with rotations on the roll axis.

Returns :

a floating-point number corresponding to the x component of the quaternion.

gyro→get_quaternionY()**YGyro****gyro→quaternionY()****gyro→get_quaternionY()**

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
double get_quaternionY( )
```

The y component is mostly correlated with rotations on the pitch axis.

Returns :

a floating-point number corresponding to the y component of the quaternion.

gyro→get_quaternionZ()

YGyro

gyro→quaternionZ()**gyro→get_quaternionZ()**

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double get_quaternionZ()

The x component is mostly correlated with changes of heading.

Returns :

a floating-point number corresponding to the z component of the quaternion.

gyro→get_recordedData()**YGyro****gyro→recordedData()****gyro→get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

`YDataSet get_recordedData(s64 startTime, s64 endTime)`

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

gyro→get_reportFrequency()
gyro→reportFrequency()
get_reportFrequency()

YGyro

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

`string get_reportFrequency()`

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

gyro→get_resolution()**YGyro****gyro→resolution()****gyro→get_resolution()**

Returns the resolution of the measured values.

`double get_resolution()`

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

gyro→get_roll()

YGyro

gyro→roll()**gyro→get_roll()**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

double **get_roll()**

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to roll angle in degrees, between -180 and +180.

gyro→get_unit()**YGyro****gyro→unit()****gyro→get_unit()**

Returns the measuring unit for the angular velocity.

string **get_unit()**

Returns :

a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns Y_UNIT_INVALID.

gyro→get_userData()

YGyro

gyro→userData()**gyro→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

`void * get_userData()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

gyro→get_xValue()**YGyro****gyro→xValue()****gyro→get_xValue()**

Returns the angular velocity around the X axis of the device, as a floating point number.

double **get_xValue()**

Returns :

a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

gyro→get_yValue()

YGyro

gyro→yValue()**gyro→get_yValue()**

Returns the angular velocity around the Y axis of the device, as a floating point number.

double **get_yValue()**

Returns :

a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns Y_YVALUE_INVALID.

gyro→get_zValue()**YGyro****gyro→zValue()****gyro→get_zValue()**

Returns the angular velocity around the Z axis of the device, as a floating point number.

double **get_zValue()**

Returns :

a floating point number corresponding to the angular velocity around the Z axis of the device, as a floating point number

On failure, throws an exception or returns Y_ZVALUE_INVALID.

gyro→isOnline()**gyro→isOnline()**

YGyro

Checks if the gyroscope is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

Returns :

`true` if the gyroscope can be reached, and `false` otherwise

gyro→load()**gyro→load()****YGyro**

Preloads the gyroscope cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

gyro→loadCalibrationPoints()**gyro→**
loadCalibrationPoints()

YGyro

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→nextGyro()**gyro→nextGyro()****YGyro**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

`YGyro * nextGyro()`

Returns :

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.

gyro→registerAnglesCallback()**gyro→**
registerAnglesCallback()

YGyro

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
int registerAnglesCallback( YAnglesCallback callback)
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGyro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

gyro→registerQuaternionCallback()**gyro→**
registerQuaternionCallback()

YGyro

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
int registerQuaternionCallback( YQuatCallback callback)
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

gyro→registerTimedReportCallback()**gyro→**
registerTimedReportCallback()**YGyro**

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YGyroTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

gyro→registerValueCallback()**gyro→**
registerValueCallback()

YGyro

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YGyroValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

gyro→**set_highestValue()****YGyro****gyro**→**setHighestValue()****gyro**→**set_highestValue()**

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logFrequency()
gyro→setLogFrequency()
set_logFrequency()

YGyro

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logicalName()**YGyro****gyro→setLogicalName()****gyro→set_logicalName()**

Changes the logical name of the gyroscope.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the gyroscope.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

gyro→set_lowestValue()**YGyro****gyro→setLowestValue()****gyro→set_lowestValue()**

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_reportFrequency()**YGyro****gyro→setReportFrequency()****gyro→****set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_resolution()**YGyro****gyro→setResolution()****gyro→set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_userdata()

YGyro

gyro→setUserData()**gyro→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.19. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_hubport.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YHubPort = yoctolib.YHubPort;
php	require_once('yocto_hubport.php');
c++	#include "yocto_hubport.h"
m	#import "yocto_hubport.h"
pas	uses yocto_hubport;
vb	yocto_hubport.vb
cs	yocto_hubport.cs
java	import com.yoctopuce.YoctoAPI.YHubPort;
py	from yocto_hubport import *

Global functions

yFindHubPort(func)

Retrieves a Yocto-hub port for a given identifier.

yFirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

YHubPort methods

hubport→describe()

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

hubport→get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

hubport→get_baudRate()

Returns the current baud rate used by this Yocto-hub port, in kbps.

hubport→get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

hubport→get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

hubport→get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

hubport→get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME . FUNCTION_NAME`.

hubport→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

hubport→get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

hubport→get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL . FUNCTIONID`.

hubport→get_logicalName()

Returns the logical name of the Yocto-hub port.

hubport→get_module()

Gets the YModule object for the device on which the function is located.

hubport→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

hubport→get_portState()

Returns the current state of the Yocto-hub port.

hubport→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

hubport→isOnline()

Checks if the Yocto-hub port is currently reachable, without raising any error.

hubport→isOnline_async(callback, context)

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

hubport→load(msValidity)

Preloads the Yocto-hub port cache with a specified validity duration.

hubport→load_async(msValidity, callback, context)

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

hubport→nextHubPort()

Continues the enumeration of Yocto-hub ports started using yFirstHubPort().

hubport→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

hubport→set_enabled(newval)

Changes the activation of the Yocto-hub port.

hubport→set_logicalName(newval)

Changes the logical name of the Yocto-hub port.

hubport→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

hubport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHubPort.FindHubPort()**YHubPort****yFindHubPort()****yFindHubPort ()**

Retrieves a Yocto-hub port for a given identifier.

```
YHubPort* yFindHubPort( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Yocto-hub port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHubPort.isOnline()` to test if the Yocto-hub port is indeed online at a given time. In case of ambiguity when looking for a Yocto-hub port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the Yocto-hub port

Returns :

a `YHubPort` object allowing you to drive the Yocto-hub port.

YHubPort.FirstHubPort()

YHubPort

yFirstHubPort()`yFirstHubPort()`

Starts the enumeration of Yocto-hub ports currently accessible.

`YHubPort* yFirstHubPort()`

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

Returns :

a pointer to a `YHubPort` object, corresponding to the first Yocto-hub port currently online, or a `null` pointer if there are none.

hubport→describe()**hubport→describe()****YHubPort**

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the Yocto-hub port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

hubport→**get_advertisedValue()**

YHubPort

hubport→**advertisedValue()****hubport**→
get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the Yocto-hub port (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

hubport→get_baudRate()**YHubPort****hubport→baudRate()****hubport→get_baudRate()**

Returns the current baud rate used by this Yocto-hub port, in kbps.

int get_baudRate()

The default value is 1000 kbps, but a slower rate may be used if communication problems are encountered.

Returns :

an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns Y_BAUDRATE_INVALID.

hubport→get_enabled()

YHubPort

hubport→enabled()**hubport→get_enabled()**

Returns true if the Yocto-hub port is powered, false otherwise.

`Y_ENABLED_enum` **get_enabled()**

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the Yocto-hub port is powered, false otherwise

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

hubport→**get_errorMessage()****YHubPort****hubport**→**errorMessage()****hubport**→**get_errorMessage()**

Returns the error message of the latest error with the Yocto-hub port.

`string` **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the Yocto-hub port object

hubport→**get_errorType()**

YHubPort

hubport→**errorType()****hubport**→**get_errorType()**

Returns the numerical error code of the latest error with the Yocto-hub port.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the Yocto-hub port object

hubport→**get_friendlyName()****YHubPort****hubport**→**friendlyName()****hubport**→
get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the Yocto-hub port if they are defined, otherwise the serial number of the module and the hardware identifier of the Yocto-hub port (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the Yocto-hub port using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

hubport→**get_functionDescriptor()**

YHubPort

hubport→**functionDescriptor()****hubport**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

hubport→**get_functionId()****YHubPort****hubport**→**functionId()****hubport**→**get_functionId()**

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the Yocto-hub port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

hubport→**get_hardwareId()**

YHubPort

hubport→**hardwareId()****hubport**→
get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Yocto-hub port. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the Yocto-hub port (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

hubport→**get_logicalName()**
hubport→**logicalName()****hubport**→
get_logicalName()

YHubPort

Returns the logical name of the Yocto-hub port.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the Yocto-hub port. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

hubport→get_module()

YHubPort

hubport→module()**hubport→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

hubport→**get_portState()****YHubPort****hubport**→**portState()****hubport**→**get_portState()**

Returns the current state of the Yocto-hub port.

`Y_PORTSTATE_enum` **get_portState()**

Returns :

a value among `Y_PORTSTATE_OFF`, `Y_PORTSTATE_OVRLD`, `Y_PORTSTATE_ON`, `Y_PORTSTATE_RUN` and `Y_PORTSTATE_PROG` corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns `Y_PORTSTATE_INVALID`.

hubport→get_userdata()

YHubPort

hubport→userdata()hubport→get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

hubport→isOnline()**hubport→isOnline()****YHubPort**

Checks if the Yocto-hub port is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Yocto-hub port.

Returns :

`true` if the Yocto-hub port can be reached, and `false` otherwise

hubport→load()**hubport→load()****YHubPort**

Preloads the Yocto-hub port cache with a specified validity duration.

YRETCODE **load(int msValidity)**

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

hubport→nextHubPort()**hubport→nextHubPort ()****YHubPort**

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort ()`.

`YHubPort * nextHubPort ()`

Returns :

a pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a `null` pointer if there are no more Yocto-hub ports to enumerate.

hubport→**registerValueCallback()****hubport**→
registerValueCallback()

YHubPort

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YHubPortValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

hubport→**set_enabled()****YHubPort****hubport**→**setEnabled()****hubport**→**set_enabled()**

Changes the activation of the Yocto-hub port.

```
int set_enabled( Y_ENABLED_enum newval)
```

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the activation of the Yocto-hub port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→**set_logicalName()****YHubPort****hubport**→**setLogicalName()****hubport**→
set_logicalName()

Changes the logical name of the Yocto-hub port.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Yocto-hub port.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

hubport→set_userdata()**YHubPort****hubport→setUserData()****hubport→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.20. Humidity function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_humidity.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YHumidity = yoctolib.YHumidity;
php	require_once('yocto_humidity.php');
c++	#include "yocto_humidity.h"
m	#import "yocto_humidity.h"
pas	uses yocto_humidity;
vb	yocto_humidity.vb
cs	yocto_humidity.cs
java	import com.yoctopuce.YoctoAPI.YHumidity;
py	from yocto_humidity import *

Global functions

yFindHumidity(func)

Retrieves a humidity sensor for a given identifier.

yFirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

YHumidity methods

humidity→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

humidity→describe()

Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

humidity→get_advertisedValue()

Returns the current value of the humidity sensor (no more than 6 characters).

humidity→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

humidity→get_currentValue()

Returns the current measure for the humidity.

humidity→get_errorMessage()

Returns the error message of the latest error with the humidity sensor.

humidity→get_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

humidity→get_friendlyName()

Returns a global identifier of the humidity sensor in the format `MODULE_NAME . FUNCTION_NAME`.

humidity→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

humidity→get_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

humidity→get_hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form `SERIAL . FUNCTIONID`.

humidity→get_highestValue()

Returns the maximal value observed for the humidity.

humidity→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

Returns the minimal value observed for the humidity.

humidity→get_module()

Gets the YModule object for the device on which the function is located.

humidity→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

humidity→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

humidity→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

humidity→get_resolution()

Returns the resolution of the measured values.

humidity→get_unit()

Returns the measuring unit for the humidity.

humidity→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

humidity→isOnline()

Checks if the humidity sensor is currently reachable, without raising any error.

humidity→isOnline_async(callback, context)

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

humidity→load(msValidity)

Preloads the humidity sensor cache with a specified validity duration.

humidity→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

humidity→load_async(msValidity, callback, context)

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

humidity→nextHumidity()

Continues the enumeration of humidity sensors started using yFirstHumidity().

humidity→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

humidity→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

humidity→set_highestValue(newval)

Changes the recorded maximal value observed for the humidity.

humidity→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

humidity→set_logicalName(newval)

Changes the logical name of the humidity sensor.

3. Reference

humidity→**set_lowestValue**(newval)

Changes the recorded minimal value observed for the humidity.

humidity→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

humidity→**set_resolution**(newval)

Changes the resolution of the measured physical values.

humidity→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

humidity→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHumidity.FindHumidity()**YHumidity****yFindHumidity()****yFindHumidity()**

Retrieves a humidity sensor for a given identifier.

```
YHumidity* yFindHumidity( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the humidity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHumidity.isOnline()` to test if the humidity sensor is indeed online at a given time. In case of ambiguity when looking for a humidity sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the humidity sensor

Returns :

a `YHumidity` object allowing you to drive the humidity sensor.

YHumidity.FirstHumidity()

YHumidity

yFirstHumidity()**yFirstHumidity()**

Starts the enumeration of humidity sensors currently accessible.

YHumidity* **yFirstHumidity()**

Use the method `YHumidity.nextHumidity()` to iterate on next humidity sensors.

Returns :

a pointer to a `YHumidity` object, corresponding to the first humidity sensor currently online, or a `null` pointer if there are none.

humidity→calibrateFromPoints()humidity→
calibrateFromPoints()

YHumidity

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→describe()**humidity→describe()****YHumidity**

Returns a short text that describes unambiguously the instance of the humidity sensor in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the humidity sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

humidity→get_advertisedValue()**YHumidity****humidity→advertisedValue()****humidity→****get_advertisedValue()**

Returns the current value of the humidity sensor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the humidity sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

humidity→get_currentRawValue()

YHumidity

humidity→currentRawValue()humidity→

get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double get_currentRawValue()

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

humidity→get_currentValue()**YHumidity****humidity→currentValue()****humidity→**
get_currentValue()

Returns the current measure for the humidity.

`double` **get_currentValue()**

Returns :

a floating point number corresponding to the current measure for the humidity

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

humidity→get_errorMessage()

YHumidity

humidity→errorMessage()**humidity→**
get_errorMessage()

Returns the error message of the latest error with the humidity sensor.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the humidity sensor object

humidity→get_errorType()**YHumidity****humidity→errorType()****humidity→get_errorType()**

Returns the numerical error code of the latest error with the humidity sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the humidity sensor object

humidity→get_friendlyName()

YHumidity

humidity→friendlyName()
humidity→get_friendlyName()

Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the humidity sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the humidity sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the humidity sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

humidity→get_functionDescriptor()**YHumidity****humidity→functionDescriptor()humidity→
get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

humidity→get_functionId()

YHumidity

humidity→functionId()humidity→
get_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the humidity sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

humidity→get_hardwareId()**YHumidity****humidity→hardwareId()humidity→****get_hardwareId()**

Returns the unique hardware identifier of the humidity sensor in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the humidity sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the humidity sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

humidity→get_highestValue()

YHumidity

humidity→highestValue()humidity→

get_highestValue()

Returns the maximal value observed for the humidity.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the humidity

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

humidity→get_logFrequency()**YHumidity****humidity→logFrequency()****humidity→****get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
string get_logFrequency( )
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

humidity→get_logicalName()

YHumidity

humidity→logicalName()humidity→

get_logicalName()

Returns the logical name of the humidity sensor.

string get_logicalName()

Returns :

a string corresponding to the logical name of the humidity sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

humidity→get_lowestValue()**YHumidity****humidity→lowestValue()****humidity→****get_lowestValue()**

Returns the minimal value observed for the humidity.

`double` **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the humidity

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

humidity→get_module()

YHumidity

humidity→module()**humidity→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

humidity→get_recordedData()**YHumidity****humidity→recordedData()****humidity→****get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YDataSet get_recordedData( s64 startTime, s64 endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

humidity→get_reportFrequency()

YHumidity

humidity→reportFrequency()humidity→

get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string get_reportFrequency()

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

humidity→get_resolution()**YHumidity****humidity→resolution()****humidity→****get_resolution()**

Returns the resolution of the measured values.

```
double get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

humidity→get_unit()

YHumidity

humidity→unit()**humidity→get_unit()**

Returns the measuring unit for the humidity.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns Y_UNIT_INVALID.

humidity→get_userdata()**YHumidity****humidity→userData()****humidity→get_userdata()**

Returns the value of the userData attribute, as previously stored using method set_userdata.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

humidity→isOnline()**humidity→isOnline()****YHumidity**

Checks if the humidity sensor is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the humidity sensor.

Returns :

`true` if the humidity sensor can be reached, and `false` otherwise

humidity→load()humidity→load()**YHumidity**

Preloads the humidity sensor cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

humidity→loadCalibrationPoints()humidity→
loadCalibrationPoints()

YHumidity

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→nextHumidity()humidity→
nextHumidity()

YHumidity

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

`YHumidity * nextHumidity()`

Returns :

a pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a `null` pointer if there are no more humidity sensors to enumerate.

humidity→**registerTimedReportCallback()****humidity**
→**registerTimedReportCallback()**

YHumidity

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YHumidityTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

humidity→registerValueCallback()**humidity→**
registerValueCallback()

YHumidity

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YHumidityValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

humidity→**set_highestValue()****YHumidity****humidity**→**setHighestValue()****humidity**→**set_highestValue()**

Changes the recorded maximal value observed for the humidity.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the humidity

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logFrequency()**YHumidity****humidity→setLogFrequency()****set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→**set_logicalName()****YHumidity****humidity**→**setLogicalName()****humidity**→**set_logicalName()**

Changes the logical name of the humidity sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the humidity sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

humidity→**set_lowestValue()****YHumidity****humidity**→**setLowestValue()****humidity**→**set_lowestValue()**

Changes the recorded minimal value observed for the humidity.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the humidity

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_reportFrequency()**YHumidity****humidity→setReportFrequency()****humidity→****set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_resolution()**YHumidity****humidity→setResolution()****humidity→****set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_userdata()

YHumidity

humidity→setUserData()humidity→

set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.21. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_led.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YLed = yoctolib.YLed;
php	require_once('yocto_led.php');
c++	#include "yocto_led.h"
m	#import "yocto_led.h"
pas	uses yocto_led;
vb	yocto_led.vb
cs	yocto_led.cs
java	import com.yoctopuce.YoctoAPI.YLed;
py	from yocto_led import *

Global functions

yFindLed(func)

Retrieves a led for a given identifier.

yFirstLed()

Starts the enumeration of leds currently accessible.

YLed methods

led→describe()

Returns a short text that describes unambiguously the instance of the led in the form TYPE (NAME) = SERIAL . FUNCTIONID.

led→get_advertisedValue()

Returns the current value of the led (no more than 6 characters).

led→get_blinking()

Returns the current led signaling mode.

led→get_errorMessage()

Returns the error message of the latest error with the led.

led→get_errorType()

Returns the numerical error code of the latest error with the led.

led→get_friendlyName()

Returns a global identifier of the led in the format MODULE_NAME . FUNCTION_NAME.

led→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

led→get_functionId()

Returns the hardware identifier of the led, without reference to the module.

led→get_hardwareId()

Returns the unique hardware identifier of the led in the form SERIAL . FUNCTIONID.

led→get_logicalName()

Returns the logical name of the led.

led→get_luminosity()

Returns the current led intensity (in per cent).

led→get_module()

	Gets the <code>YModule</code> object for the device on which the function is located.
led→get_module_async(callback, context)	Gets the <code>YModule</code> 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 <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
led→isOnline()	Checks if the led is currently reachable, without raising any error.
led→isOnline_async(callback, context)	Checks if the led is currently reachable, without raising any error (asynchronous version).
led→load(msValidity)	Preloads the led cache with a specified validity duration.
led→load_async(msValidity, callback, context)	Preloads the led cache with a specified validity duration (asynchronous version).
led→nextLed()	Continues the enumeration of leds started using <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 <code>userData</code> attribute of the function.
led→wait_async(callback, context)	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLed.FindLed()**YLed****yFindLed()****yFindLed()**

Retrieves a led for a given identifier.

```
YLed* yFindLed( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the led is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLed.isOnline()` to test if the led is indeed online at a given time. In case of ambiguity when looking for a led by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the led

Returns :

a YLed object allowing you to drive the led.

YLed.FirstLed()

YLed

yFirstLed()`yFirstLed()`

Starts the enumeration of leds currently accessible.

`YLed* yFirstLed()`

Use the method `YLed.nextLed()` to iterate on next leds.

Returns :

a pointer to a `YLed` object, corresponding to the first led currently online, or a `null` pointer if there are none.

led→describe()**led→describe()****YLed**

Returns a short text that describes unambiguously the instance of the led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

led→**get_advertisedValue()**

YLed

led→**advertisedValue()****led**→

get_advertisedValue()

Returns the current value of the led (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the led (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

led→**get_blinking()****YLed****led**→**blinking()****led**→**get_blinking()**

Returns the current led signaling mode.

`Y_BLINKING_enum` **get_blinking()**

Returns :

a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

On failure, throws an exception or returns `Y_BLINKING_INVALID`.

led→**get_errorMessage()**

YLed

led→**errorMessage()** **led**→**get_errorMessage()**

Returns the error message of the latest error with the led.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the led object

led→get_errorType()**YLed****led**→errorType()**led**→get_errorType()

Returns the numerical error code of the latest error with the led.

YRETCODE get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the led object

led→**get_friendlyName()**

YLed

led→**friendlyName()****led**→**get_friendlyName()**

Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the led if they are defined, otherwise the serial number of the module and the hardware identifier of the led (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the led using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

led→**get_functionDescriptor()****YLed****led**→**functionDescriptor()****led**→
get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

`YFUN_DESCR` **get_functionDescriptor()**

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

led→**get_functionId()**

YLed

led→**functionId()****led**→**get_functionId()**

Returns the hardware identifier of the led, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the led (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

led→**get_hardwareId()****YLed****led**→**hardwareId()****led**→**get_hardwareId()**

Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the led. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the led (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

led→**get_logicalName()**

YLed

led→**logicalName()** **led**→**get_logicalName()**

Returns the logical name of the led.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the led. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

led→get_luminosity()**YLed****led→luminosity()****led→get_luminosity()**

Returns the current led intensity (in per cent).

int **get_luminosity**()

Returns :

an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.

led→get_module()

YLed

led→module()led→get_module()

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

led→get_power()**YLed****led→power()****led→get_power()**

Returns the current led state.

`Y_POWER_enum` **get_power()**

Returns :

either `Y_POWER_OFF` or `Y_POWER_ON`, according to the current led state

On failure, throws an exception or returns `Y_POWER_INVALID`.

led→get_userdata()

YLed

led→userData() led→get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

led→isOnline()**led→isOnline()****YLed**

Checks if the led is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the led.

Returns :

`true` if the led can be reached, and `false` otherwise

led→load()**led→load()****YLed**

Preloads the led cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

led→nextLed()led→nextLed()**YLed**

Continues the enumeration of leds started using `yFirstLed()`.

YLed * **nextLed**()

Returns :

a pointer to a YLed object, corresponding to a led currently online, or a `null` pointer if there are no more leds to enumerate.

led→registerValueCallback() led→
registerValueCallback()

YLed

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YLedValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

led→set_blinking()**YLed****led→setBlinking()****led→set_blinking()**

Changes the current led signaling mode.

```
int set_blinking( Y_BLINKING_enum newval)
```

Parameters :

newval a value among Y_BLINKING_STILL, Y_BLINKING_RELAX, Y_BLINKING_AWARE, Y_BLINKING_RUN, Y_BLINKING_CALL and Y_BLINKING_PANIC corresponding to the current led signaling mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→**set_logicalName()****YLed****led**→**setLogicalName()****led**→**set_logicalName()**

Changes the logical name of the led.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the led.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

led→**set_luminosity()****YLed****led**→**setLuminosity()****led**→**set_luminosity()**

Changes the current led intensity (in per cent).

```
int set_luminosity( int newval)
```

Parameters :

newval an integer corresponding to the current led intensity (in per cent)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→**set_power()**

YLed

led→**setPower()****led**→**set_power()**

Changes the state of the led.

```
int set_power( Y_POWER_enum newval)
```

Parameters :

newval either Y_POWER_OFF or Y_POWER_ON, according to the state of the led

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→set_userdata()**YLed****led→setUserData()****led→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.22. LightSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_lightsensor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YLightSensor = yoctolib.YLightSensor;
php	require_once('yocto_lightsensor.php');
c++	#include "yocto_lightsensor.h"
m	#import "yocto_lightsensor.h"
pas	uses yocto_lightsensor;
vb	yocto_lightsensor.vb
cs	yocto_lightsensor.cs
java	import com.yoctopuce.YoctoAPI.YLightSensor;
py	from yocto_lightsensor import *

Global functions

yFindLightSensor(func)

Retrieves a light sensor for a given identifier.

yFirstLightSensor()

Starts the enumeration of light sensors currently accessible.

YLightSensor methods

lightsensor→calibrate(calibratedVal)

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

lightsensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

lightsensor→describe()

Returns a short text that describes unambiguously the instance of the light sensor in the form TYPE (NAME) =SERIAL . FUNCTIONID.

lightsensor→get_advertisedValue()

Returns the current value of the light sensor (no more than 6 characters).

lightsensor→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

lightsensor→get_currentValue()

Returns the current measure for the ambient light.

lightsensor→get_errorMessage()

Returns the error message of the latest error with the light sensor.

lightsensor→get_errorType()

Returns the numerical error code of the latest error with the light sensor.

lightsensor→get_friendlyName()

Returns a global identifier of the light sensor in the format MODULE_NAME . FUNCTION_NAME.

lightsensor→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

lightsensor→get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

lightsensor→get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

lightsensor→get_highestValue()

Returns the maximal value observed for the ambient light.

lightsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

lightsensor→get_logicalName()

Returns the logical name of the light sensor.

lightsensor→get_lowestValue()

Returns the minimal value observed for the ambient light.

lightsensor→get_module()

Gets the `YModule` object for the device on which the function is located.

lightsensor→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

lightsensor→get_recordedData(startTime, endTime)

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

lightsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

lightsensor→get_resolution()

Returns the resolution of the measured values.

lightsensor→get_unit()

Returns the measuring unit for the ambient light.

lightsensor→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

lightsensor→isOnline()

Checks if the light sensor is currently reachable, without raising any error.

lightsensor→isOnline_async(callback, context)

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

lightsensor→load(msValidity)

Preloads the light sensor cache with a specified validity duration.

lightsensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

lightsensor→load_async(msValidity, callback, context)

Preloads the light sensor cache with a specified validity duration (asynchronous version).

lightsensor→nextLightSensor()

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

lightsensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

lightsensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

lightsensor→set_highestValue(newval)

Changes the recorded maximal value observed for the ambient light.

lightsensor→set_logFrequency(newval)

3. Reference

Changes the datalogger recording frequency for this function.

lightsensor→**set_logicalName**(newval)

Changes the logical name of the light sensor.

lightsensor→**set_lowestValue**(newval)

Changes the recorded minimal value observed for the ambient light.

lightsensor→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

lightsensor→**set_resolution**(newval)

Changes the resolution of the measured physical values.

lightsensor→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

lightsensor→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLightSensor.FindLightSensor() yFindLightSensor()yFindLightSensor()

YLightSensor

Retrieves a light sensor for a given identifier.

```
YLightSensor* yFindLightSensor( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the light sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLightSensor.isOnline()` to test if the light sensor is indeed online at a given time. In case of ambiguity when looking for a light sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the light sensor

Returns :

a `YLightSensor` object allowing you to drive the light sensor.

YLightSensor.FirstLightSensor()

YLightSensor

yFirstLightSensor()`yFirstLightSensor()`

Starts the enumeration of light sensors currently accessible.

`YLightSensor* yFirstLightSensor()`

Use the method `YLightSensor.nextLightSensor()` to iterate on next light sensors.

Returns :

a pointer to a `YLightSensor` object, corresponding to the first light sensor currently online, or a `null` pointer if there are none.

lightsensor→calibrate()**lightsensor→calibrate()****YLightSensor**

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```
int calibrate( double calibratedVal)
```

Parameters :

calibratedVal the desired target value.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→calibrateFromPoints()lightsensor→
calibrateFromPoints()

YLightSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→describe()**lightsensor→describe()****YLightSensor**

Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the light sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

lightsensor→get_advertisedValue()

YLightSensor

lightsensor→advertisedValue()lightsensor→
get_advertisedValue()

Returns the current value of the light sensor (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the light sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

lightsensor→get_currentRawValue()**YLightSensor****lightsensor→currentRawValue()****lightsensor→**
get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

lightsensor→get_currentValue()

YLightSensor

lightsensor→currentValue()**lightsensor→**
get_currentValue()

Returns the current measure for the ambient light.

`double get_currentValue()`

Returns :

a floating point number corresponding to the current measure for the ambient light

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

lightsensor→get_errorMessage()**YLightSensor****lightsensor→errorMessage()****lightsensor→**
get_errorMessage()

Returns the error message of the latest error with the light sensor.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the light sensor object

lightsensor→get_errorType()

YLightSensor

lightsensor→errorType()lightsensor→

get_errorType()

Returns the numerical error code of the latest error with the light sensor.

[YRETCODE](#) **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the light sensor object

lightsensor→get_friendlyName()**YLightSensor****lightsensor→friendlyName()****lightsensor→**
get_friendlyName()

Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the light sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the light sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the light sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

lightsensor→get_functionDescriptor()

YLightSensor

**lightsensor→functionDescriptor()lightsensor→
get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

lightsensor→get_functionId()**YLightSensor****lightsensor→functionId()****lightsensor→**
get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the light sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

lightsensor→get_hardwareId()

YLightSensor

lightsensor→hardwareId()**lightsensor→**
get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the light sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the light sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

lightsensor→**get_highestValue()****YLightSensor****lightsensor**→**highestValue()****lightsensor**→
get_highestValue()

Returns the maximal value observed for the ambient light.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the ambient light

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

lightsensor→get_logFrequency()**YLightSensor****lightsensor→logFrequency()****lightsensor→**
get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string get_logFrequency()**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

lightsensor→**get_logicalName()****YLightSensor****lightsensor**→**logicalName()****lightsensor**→
get_logicalName()

Returns the logical name of the light sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the light sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

lightsensor→get_lowestValue()

YLightSensor

lightsensor→lowestValue()**lightsensor→**
get_lowestValue()

Returns the minimal value observed for the ambient light.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the ambient light

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

lightsensor→get_module()**YLightSensor****lightsensor→module()****lightsensor→**
get_module()

Gets the YModule object for the device on which the function is located.

```
YModule * get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

lightsensor→**get_recordedData()****YLightSensor****lightsensor**→**recordedData()****lightsensor**→
get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

lightsensor→get_reportFrequency()**YLightSensor****lightsensor→reportFrequency()****lightsensor→**
get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
string get_reportFrequency( )
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

lightsensor→get_resolution()

YLightSensor

lightsensor→resolution()**lightsensor→**
get_resolution()

Returns the resolution of the measured values.

double **get_resolution()**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

lightsensor→get_unit()**YLightSensor****lightsensor→unit()****lightsensor→get_unit()**

Returns the measuring unit for the ambient light.

string **get_unit()**

Returns :

a string corresponding to the measuring unit for the ambient light

On failure, throws an exception or returns Y_UNIT_INVALID.

lightsensor→get_userdata()

YLightSensor

lightsensor→userdata()**lightsensor→**
get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

lightsensor→isOnline()**lightsensor→isOnline()****YLightSensor**

Checks if the light sensor is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

Returns :

`true` if the light sensor can be reached, and `false` otherwise

lightsensor→load()**lightsensor→load()****YLightSensor**

Preloads the light sensor cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

lightsensor→loadCalibrationPoints()lightsensor→
loadCalibrationPoints()

YLightSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→**nextLightSensor()****lightsensor**→
nextLightSensor()

YLightSensor

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

`YLightSensor *` **nextLightSensor()**

Returns :

a pointer to a `YLightSensor` object, corresponding to a light sensor currently online, or a `null` pointer if there are no more light sensors to enumerate.

lightsensor→registerTimedReportCallback()**YLightSensor****lightsensor→registerTimedReportCallback()**

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YLightSensorTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

lightsensor→registerValueCallback()lightsensor→
registerValueCallback()

YLightSensor

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YLightSensorValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

lightsensor→**set_highestValue()****YLightSensor****lightsensor**→**setHighestValue()****lightsensor**→
set_highestValue()

Changes the recorded maximal value observed for the ambient light.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the ambient light

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logFrequency()**YLightSensor****lightsensor→setLogFrequency()****lightsensor→set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logicalName()**YLightSensor****lightsensor→setLogicalName()****lightsensor→**
set_logicalName()

Changes the logical name of the light sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the light sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

lightsensor→**set_lowestValue()****YLightSensor****lightsensor**→**setLowestValue()****lightsensor**→**set_lowestValue()**

Changes the recorded minimal value observed for the ambient light.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the ambient light

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_reportFrequency()**YLightSensor****lightsensor→setReportFrequency()****lightsensor→**
set_reportFrequency()

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_resolution()**YLightSensor****lightsensor→setResolution()****lightsensor→**
set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_userdata()**YLightSensor****lightsensor→setUserData()****lightsensor→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.23. Magnetometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_magnetometer.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YMagnetometer = yoctolib.YMagnetometer;
php	require_once('yocto_magnetometer.php');
c++	#include "yocto_magnetometer.h"
m	#import "yocto_magnetometer.h"
pas	uses yocto_magnetometer;
vb	yocto_magnetometer.vb
cs	yocto_magnetometer.cs
java	import com.yoctopuce.YoctoAPI.YMagnetometer;
py	from yocto_magnetometer import *

Global functions

yFindMagnetometer(func)

Retrieves a magnetometer for a given identifier.

yFirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

YMagnetometer methods

magnetometer→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

magnetometer→describe()

Returns a short text that describes unambiguously the instance of the magnetometer in the form TYPE (NAME) = SERIAL . FUNCTIONID.

magnetometer→get_advertisedValue()

Returns the current value of the magnetometer (no more than 6 characters).

magnetometer→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

magnetometer→get_currentValue()

Returns the current value of the magnetic field.

magnetometer→get_errorMessage()

Returns the error message of the latest error with the magnetometer.

magnetometer→get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

magnetometer→get_friendlyName()

Returns a global identifier of the magnetometer in the format MODULE_NAME . FUNCTION_NAME.

magnetometer→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

magnetometer→get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

magnetometer→get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form SERIAL . FUNCTIONID.

magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

magnetometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

magnetometer→get_logicalName()

Returns the logical name of the magnetometer.

magnetometer→get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

magnetometer→get_module()

Gets the YModule object for the device on which the function is located.

magnetometer→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

magnetometer→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

magnetometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

magnetometer→get_resolution()

Returns the resolution of the measured values.

magnetometer→get_unit()

Returns the measuring unit for the magnetic field.

magnetometer→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

magnetometer→get_xValue()

Returns the X component of the magnetic field, as a floating point number.

magnetometer→get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

magnetometer→get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

magnetometer→isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

magnetometer→isOnline_async(callback, context)

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

magnetometer→load(msValidity)

Preloads the magnetometer cache with a specified validity duration.

magnetometer→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

magnetometer→load_async(msValidity, callback, context)

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

magnetometer→nextMagnetometer()

Continues the enumeration of magnetometers started using yFirstMagnetometer().

magnetometer→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

magnetometer→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

3. Reference

magnetometer→set_highestValue(newval)

Changes the recorded maximal value observed.

magnetometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

magnetometer→set_logicalName(newval)

Changes the logical name of the magnetometer.

magnetometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

magnetometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

magnetometer→set_resolution(newval)

Changes the resolution of the measured physical values.

magnetometer→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

magnetometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMagnetometer.FindMagnetometer() yFindMagnetometer()yFindMagnetometer()

YMagnetometer

Retrieves a magnetometer for a given identifier.

```
YMagnetometer* yFindMagnetometer( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMagnetometer.isOnline()` to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the magnetometer

Returns :

a `YMagnetometer` object allowing you to drive the magnetometer.

YMagnetometer.FirstMagnetometer()

YMagnetometer

yFirstMagnetometer()`yFirstMagnetometer()`

Starts the enumeration of magnetometers currently accessible.

`YMagnetometer*` **yFirstMagnetometer()**

Use the method `YMagnetometer.nextMagnetometer()` to iterate on next magnetometers.

Returns :

a pointer to a `YMagnetometer` object, corresponding to the first magnetometer currently online, or a `null` pointer if there are none.

magnetometer→calibrateFromPoints()**YMagnetometer****magnetometer→calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→**describe()****magnetometer**→
describe()

YMagnetometer

Returns a short text that describes unambiguously the instance of the magnetometer in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the magnetometer (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

magnetometer→**get_advertisedValue()****YMagnetometer****magnetometer**→**advertisedValue()****magnetometer**→
get_advertisedValue()

Returns the current value of the magnetometer (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the magnetometer (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

magnetometer→get_currentRawValue()

YMagnetometer

magnetometer→currentRawValue()
magnetometer→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double get_currentRawValue()

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

magnetometer→**get_currentValue()****YMagnetometer****magnetometer**→**currentValue()****magnetometer**→
get_currentValue()

Returns the current value of the magnetic field.

double **get_currentValue()**

Returns :

a floating point number corresponding to the current value of the magnetic field

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

magnetometer→get_errorMessage()

YMagnetometer

magnetometer→errorMessage()**magnetometer→**
get_errorMessage()

Returns the error message of the latest error with the magnetometer.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the magnetometer object

magnetometer→get_errorType()**YMagnetometer****magnetometer→errorType()****magnetometer→**
get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the magnetometer object

magnetometer→**get_friendlyName()**

YMagnetometer

magnetometer→**friendlyName()****magnetometer**→
get_friendlyName()

Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the magnetometer if they are defined, otherwise the serial number of the module and the hardware identifier of the magnetometer (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the magnetometer using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

magnetometer→**get_functionDescriptor()****YMagnetometer****magnetometer**→**functionDescriptor()****magnetometer**
→**get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

magnetometer→**get_functionId()**

YMagnetometer

magnetometer→**functionId()****magnetometer**→
get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the magnetometer (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

magnetometer→get_hardwareId()**YMagnetometer****magnetometer→hardwareId()****magnetometer→**
get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the magnetometer. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the magnetometer (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

magnetometer→get_highestValue()

YMagnetometer

magnetometer→highestValue()
magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

double get_highestValue()

Returns :

a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

magnetometer→get_logFrequency()**YMagnetometer****magnetometer→logFrequency()****magnetometer→**
get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

magnetometer→**get_logicalName()**

YMagnetometer

magnetometer→**logicalName()****magnetometer**→
get_logicalName()

Returns the logical name of the magnetometer.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the magnetometer. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

magnetometer→**get_lowestValue()****YMagnetometer****magnetometer**→**lowestValue()****magnetometer**→
get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

`double` **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

magnetometer→**get_module()**

YMagnetometer

magnetometer→**module()****magnetometer**→
get_module()

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

magnetometer→**get_recordedData()****YMagnetometer****magnetometer**→**recordedData()****magnetometer**→
get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

magnetometer→get_reportFrequency()

YMagnetometer

magnetometer→reportFrequency()magnetometer→

get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string get_reportFrequency()

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

magnetometer→**get_resolution()****YMagnetometer****magnetometer**→**resolution()****magnetometer**→
get_resolution()

Returns the resolution of the measured values.

`double` **get_resolution()** `()`

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

magnetometer→get_unit()

YMagnetometer

magnetometer→unit()~~magnetometer→get_unit()~~

Returns the measuring unit for the magnetic field.

string **get_unit()**

Returns :

a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns Y_UNIT_INVALID.

magnetometer→get_userData()**YMagnetometer****magnetometer→userData()****magnetometer→**
get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
void * get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

magnetometer→get_xValue()

YMagnetometer

magnetometer→xValue()**magnetometer→**
get_xValue()

Returns the X component of the magnetic field, as a floating point number.

`double get_xValue()`

Returns :

a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

magnetometer→**get_yValue()****YMagnetometer****magnetometer**→**yValue()****magnetometer**→**get_yValue()**

Returns the Y component of the magnetic field, as a floating point number.

double **get_yValue()**

Returns :

a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_YVALUE_INVALID.

magnetometer→**get_zValue()**

YMagnetometer

magnetometer→**zValue()****magnetometer**→
get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

`double get_zValue()`

Returns :

a floating point number corresponding to the Z component of the magnetic field, as a floating point number

On failure, throws an exception or returns `Y_ZVALUE_INVALID`.

magnetometer→**isOnline()****magnetometer**→
isOnline()

YMagnetometer

Checks if the magnetometer is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the magnetometer.

Returns :

`true` if the magnetometer can be reached, and `false` otherwise

magnetometer→load()**magnetometer→load()****YMagnetometer**

Preloads the magnetometer cache with a specified validity duration.

YRETCODE **load(int msValidity)**

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

magnetometer→loadCalibrationPoints()**YMagnetometer****magnetometer→loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→**nextMagnetometer()****magnetometer**
→**nextMagnetometer()**

YMagnetometer

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

`YMagnetometer * nextMagnetometer()`

Returns :

a pointer to a `YMagnetometer` object, corresponding to a magnetometer currently online, or a `null` pointer if there are no more magnetometers to enumerate.

magnetometer→registerTimedReportCallback()**YMagnetometer****magnetometer→****registerTimedReportCallback()**

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YMagnetometerTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

magnetometer→registerValueCallback()**YMagnetometer****magnetometer→registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YMagnetometerValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

magnetometer→set_highestValue()**YMagnetometer****magnetometer→setHighestValue()****magnetometer→**
set_highestValue()

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logFrequency()

YMagnetometer

magnetometer→setLogFrequency()**magnetometer→set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logicalName()**YMagnetometer****magnetometer→setLogicalName()****magnetometer→set_logicalName()**

Changes the logical name of the magnetometer.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the magnetometer.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

magnetometer→set_lowestValue()

YMagnetometer

magnetometer→setLowestValue()
magnetometer→set_lowestValue()

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_reportFrequency()**YMagnetometer****magnetometer→setReportFrequency()****magnetometer→set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→**set_resolution()****YMagnetometer****magnetometer**→**setResolution()****magnetometer**→
set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_userdata()**YMagnetometer****magnetometer→setUserData()****magnetometer→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.24. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

YMeasure methods

measure→get_averageValue()

Returns the average value observed during the time interval covered by this measure.

measure→get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_maxValue()

Returns the largest value observed during the time interval covered by this measure.

measure→get_minValue()

Returns the smallest value observed during the time interval covered by this measure.

measure→get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→**get_averageValue()****YMeasure****measure**→**averageValue()****measure**→
get_averageValue()

Returns the average value observed during the time interval covered by this measure.

`double` **get_averageValue()**

Returns :

a floating-point number corresponding to the average value observed.

measure→**get_endTimeUTC()**

YMeasure

measure→**endTimeUTC()****measure**→
get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

`double get_endTimeUTC()`

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→get_maxValue()**YMeasure****measure→maxValue()****measure→get_maxValue()**

Returns the largest value observed during the time interval covered by this measure.

double **get_maxValue()**

Returns :

a floating-point number corresponding to the largest value observed.

measure→**get_minValue()**

YMeasure

measure→**minValue()****measure**→**get_minValue()**

Returns the smallest value observed during the time interval covered by this measure.

double **get_minValue()**

Returns :

a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC()**YMeasure****measure→startTimeUTC()****measure→**
get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

double **get_startTimeUTC()**

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

3.25. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

Global functions
yFindModule(func) Allows you to find a module from its serial number or from its logical name.
yFirstModule() Starts the enumeration of modules currently accessible.
YModule methods
module→describe() Returns a descriptive text that identifies the module.
module→download(pathname) Downloads the specified built-in file and returns a binary buffer with its content.
module→functionCount() Returns the number of functions (beside the "module" interface) available on the module.
module→functionId(functionIndex) Retrieves the hardware identifier of the <i>n</i> th function on the module.
module→functionName(functionIndex) Retrieves the logical name of the <i>n</i> th function on the module.
module→functionValue(functionIndex) Retrieves the advertised value of the <i>n</i> th function on the module.
module→get_beacon() Returns the state of the localization beacon.
module→get_errorMessage() Returns the error message of the latest error with this module object.
module→get_errorType() Returns the numerical error code of the latest error with this module object.
module→get_firmwareRelease() Returns the version of the firmware embedded in the module.
module→get_hardwareId() Returns the unique hardware identifier of the module.
module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()

Returns a string with last logs of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

Returns the luminosity of the module informative leds (from 0 to 100).

module→get_persistentSettings()

Returns the current state of persistent module settings.

module→get_productId()

Returns the USB device identifier of the module.

module→get_productName()

Returns the commercial name of the module, as set by the factory.

module→get_productRelease()

Returns the hardware release version of the module.

module→get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()

Returns the serial number of the module, as set by the factory.

module→get_upTime()

Returns the number of milliseconds spent since the module was powered on.

module→get_usbBandwidth()

Returns the number of USB interfaces used by the module.

module→get_usbCurrent()

Returns the current consumed by the module on the USB bus, in milli-amps.

module→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

module→isOnline()

Checks if the module is currently reachable, without raising any error.

module→isOnline_async(callback, context)

Checks if the module is currently reachable, without raising any error.

module→load(msValidity)

Preloads the module cache with a specified validity duration.

module→load_async(msValidity, callback, context)

Preloads the module cache with a specified validity duration (asynchronous version).

module→nextModule()

Continues the module enumeration started using `yFirstModule()`.

module→reboot(secBeforeReboot)

Schedules a simple module reboot after the given number of seconds.

module→registerLogCallback(callback)

todo

module→revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

3. Reference

module→**set_beacon**(**newval**)

Turns on or off the module localization beacon.

module→**set_logicalName**(**newval**)

Changes the logical name of the module.

module→**set_luminosity**(**newval**)

Changes the luminosity of the module informative leds.

module→**set_usbBandwidth**(**newval**)

Changes the number of USB interfaces used by the module.

module→**set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

module→**triggerFirmwareUpdate**(**secBeforeReboot**)

Schedules a module reboot into special firmware update mode.

module→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YModule.FindModule()**YModule****yFindModule()****yFindModule()**

Allows you to find a module from its serial number or from its logical name.

```
YModule* yFindModule( string func)
```

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string containing either the serial number or the logical name of the desired module

Returns :

a `YModule` object allowing you to drive the module or get additional information on the module.

YModule.FirstModule()

YModule

yFirstModule()**yFirstModule()**

Starts the enumeration of modules currently accessible.

YModule* **yFirstModule()**

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

module→**describe()****module**→**describe()****YModule**

Returns a descriptive text that identifies the module.

string **describe()**

The text may include either the logical name or the serial number of the module.

Returns :

a string that describes the module

module→**download()****module**→**download()**

YModule

Downloads the specified built-in file and returns a binary buffer with its content.

```
string download( string pathname)
```

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

module→**functionCount()****module**→
functionCount()

YModule

Returns the number of functions (beside the "module" interface) available on the module.

int functionCount()

Returns :

the number of functions on the module

On failure, throws an exception or returns a negative error code.

module→**functionId()****module**→**functionId()****YModule**

Retrieves the hardware identifier of the *n*th function on the module.

```
string functionId( int functionIndex)
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

module→**functionName()****module**→**functionName()****YModule**

Retrieves the logical name of the *n*th function on the module.

```
string functionName( int functionIndex)
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

module→**functionValue()****module**→
functionValue()

YModule

Retrieves the advertised value of the n th function on the module.

```
string functionValue( int functionIndex)
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

module→get_beacon()**YModule****module→beacon()****module→get_beacon()**

Returns the state of the localization beacon.

[Y_BEACON_enum](#) **get_beacon()**

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module→**get_errorMessage()**

YModule

module→**errorMessage()****module**→

get_errorMessage()

Returns the error message of the latest error with this module object.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this module object

module→**get_errorType()****YModule****module**→**errorType()****module**→**get_errorType()**

Returns the numerical error code of the latest error with this module object.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using this module object

module→get_firmwareRelease()

YModule

module→firmwareRelease()**module→**

get_firmwareRelease()

Returns the version of the firmware embedded in the module.

string get_firmwareRelease()

Returns :

a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns Y_FIRMWARERELEASE_INVALID.

module→**get_hardwareId()****YModule****module**→**hardwareId()****module**→**get_hardwareId()**

Returns the unique hardware identifier of the module.

string **get_hardwareId()**

The unique hardware identifier is made of the device serial number followed by string ".module".

Returns :

a string that uniquely identifies the module

module→**get_icon2d()**

YModule

module→**icon2d()****module**→**get_icon2d()**

Returns the icon of the module.

string **get_icon2d()**

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format.

module→**get_lastLogs()****YModule****module**→**lastLogs()****module**→**get_lastLogs()**

Returns a string with last logs of the module.

string **get_lastLogs()**

This method return only logs that are still in the module.

Returns :

a string with last logs of the module.

module→**get_logicalName()**

YModule

module→**logicalName()****module**→

get_logicalName()

Returns the logical name of the module.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the module

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

module→get_luminosity()**YModule****module→luminosity()****module→get_luminosity()**

Returns the luminosity of the module informative leds (from 0 to 100).

int **get_luminosity()**

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.

module→**get_persistentSettings()**
module→**persistentSettings()****module**→
get_persistentSettings()

YModule

Returns the current state of persistent module settings.

[Y_PERSISTENTSETTINGS_enum](#) **get_persistentSettings()**

Returns :

a value among `Y_PERSISTENTSETTINGS_LOADED`, `Y_PERSISTENTSETTINGS_SAVED` and `Y_PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `Y_PERSISTENTSETTINGS_INVALID`.

module→**get_productId()****YModule****module**→**productId()****module**→**get_productId()**

Returns the USB device identifier of the module.

int **get_productId()**

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y_PRODUCTID_INVALID.

module→**get_productName()**

YModule

module→**productName()****module**→

get_productName()

Returns the commercial name of the module, as set by the factory.

string **get_productName()**

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y_PRODUCTNAME_INVALID.

module→**get_productRelease()****YModule****module**→**productRelease()****module**→**get_productRelease()**

Returns the hardware release version of the module.

int **get_productRelease()**

Returns :

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.

module→get_rebootCountdown()
module→rebootCountdown()
get_rebootCountdown()

YModule

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()**
module→**serialNumber()****module**→
get_serialNumber()

YModule

Returns the serial number of the module, as set by the factory.

string **get_serialNumber()**

Returns :

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns Y_SERIALNUMBER_INVALID.

module→get_upTime()

YModule

module→upTime()**module→get_upTime()**

Returns the number of milliseconds spent since the module was powered on.

s64 **get_upTime()**

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns Y_UPTIME_INVALID.

module→**get_usbBandwidth()****YModule****module**→**usbBandwidth()****module**→**get_usbBandwidth()**

Returns the number of USB interfaces used by the module.

[Y_USBBANDWIDTH_enum](#) **get_usbBandwidth()**

Returns :

either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

On failure, throws an exception or returns `Y_USBBANDWIDTH_INVALID`.

module→get_usbCurrent()

YModule

module→usbCurrent()**module→get_usbCurrent()**

Returns the current consumed by the module on the USB bus, in milli-amps.

int get_usbCurrent()

Returns :

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns Y_USBCURRENT_INVALID.

module→get_userdata()**YModule****module→userdata()****module→get_userdata()**

Returns the value of the userdata attribute, as previously stored using method `set_userdata`.

```
void * get_userdata()
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

module→**isOnline()****module**→**isOnline()**

YModule

Checks if the module is currently reachable, without raising any error.

bool **isOnline()**

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

`true` if the module can be reached, and `false` otherwise

module→load()**module→load()****YModule**

Preloads the module cache with a specified validity duration.

YRETCODE load(int msValidity)

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→**nextModule()****module**→**nextModule()**

YModule

Continues the module enumeration started using `yFirstModule()`.

`YModule * nextModule()`

Returns :

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot()**module→reboot ()****YModule**

Schedules a simple module reboot after the given number of seconds.

```
int reboot( int secBeforeReboot)
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→**registerLogCallback()****module**→
registerLogCallback()

YModule

todo

```
void registerLogCallback( YModuleLogCallback callback)
```

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.

module→**revertFromFlash()****module**→
revertFromFlash()

YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

int **revertFromFlash()**

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→saveToFlash()**module→saveToFlash()**

YModule

Saves current settings in the nonvolatile memory of the module.

int saveToFlash()

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→**set_beacon()****YModule****module**→**setBeacon()****module**→**set_beacon()**

Turns on or off the module localization beacon.

```
int set_beacon( Y_BEACON_enum newval)
```

Parameters :

newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_logicalName()****YModule****module**→**setLogicalName()****module**→**set_logicalName()**

Changes the logical name of the module.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_luminosity()****YModule****module**→**setLuminosity()****module**→
set_luminosity()

Changes the luminosity of the module informative leds.

```
int set_luminosity( int newval)
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_usbBandwidth()****YModule****module**→**setUsbBandwidth()****module**→**set_usbBandwidth()**

Changes the number of USB interfaces used by the module.

```
int set_usbBandwidth( Y_USBBANDWIDTH_enum newval)
```

You must reboot the module after changing this setting.

Parameters :

newval either Y_USBBANDWIDTH_SIMPLE or Y_USBBANDWIDTH_DOUBLE, according to the number of USB interfaces used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_userdata()**YModule****module→setUserData()****module→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* 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()****module**→
triggerFirmwareUpdate()

YModule

Schedules a module reboot into special firmware update mode.

```
int triggerFirmwareUpdate( int secBeforeReboot)
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

3.26. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_network.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YNetwork = yoctolib.YNetwork;
php	require_once('yocto_network.php');
cpp	#include "yocto_network.h"
m	#import "yocto_network.h"
pas	uses yocto_network;
vb	yocto_network.vb
cs	yocto_network.cs
java	import com.yoctopuce.YoctoAPI.YNetwork;
py	from yocto_network import *

Global functions

yFindNetwork(func)

Retrieves a network interface for a given identifier.

yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

YNetwork methods

network→callbackLogin(username, password)

Connects to the notification callback and saves the credentials required to log into it.

network→describe()

Returns a short text that describes unambiguously the instance of the network interface in the form TYPE (NAME) = SERIAL . FUNCTIONID.

network→get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

network→get_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

network→get_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

network→get_callbackEncoding()

Returns the encoding standard to use for representing notification values.

network→get_callbackMaxDelay()

Returns the maximum waiting time between two callback notifications, in seconds.

network→get_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

network→get_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

network→get_callbackUrl()

Returns the callback URL to notify of significant state changes.

network→get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→get_errorMessage()

Returns the error message of the latest error with the network interface.

network→get_errorType()

Returns the numerical error code of the latest error with the network interface.

network→get_friendlyName()

Returns a global identifier of the network interface in the format `MODULE_NAME . FUNCTION_NAME`.

network→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

network→get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

network→get_hardwareId()

Returns the unique hardware identifier of the network interface in the form `SERIAL . FUNCTIONID`.

network→get_ipAddress()

Returns the IP address currently in use by the device.

network→get_logicalName()

Returns the logical name of the network interface.

network→get_macAddress()

Returns the MAC address of the network interface.

network→get_module()

Gets the `YModule` object for the device on which the function is located.

network→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

network→get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

network→get_primaryDNS()

Returns the IP address of the primary name server to be used by the module.

network→get_readiness()

Returns the current established working mode of the network interface.

network→get_router()

Returns the IP address of the router on the device subnet (default gateway).

network→get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

network→get_subnetMask()

Returns the subnet mask currently used by the device.

network→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

network→get_userPassword()

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

network→get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→isOnline()

Checks if the network interface is currently reachable, without raising any error.

network→isOnline_async(callback, context)

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

network→load(msValidity)

Preloads the network interface cache with a specified validity duration.

network→load_async(msValidity, callback, context)

Preloads the network interface cache with a specified validity duration (asynchronous version).

network→nextNetwork()

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

network→ping(host)

Pings `str_host` to test the network connectivity.

network→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

network→set_adminPassword(newval)

Changes the password for the "admin" user.

network→set_callbackCredentials(newval)

Changes the credentials required to connect to the callback address.

network→set_callbackEncoding(newval)

Changes the encoding standard to use for representing notification values.

network→set_callbackMaxDelay(newval)

Changes the maximum waiting time between two callback notifications, in seconds.

network→set_callbackMethod(newval)

Changes the HTTP method used to notify callbacks for significant state changes.

network→set_callbackMinDelay(newval)

Changes the minimum waiting time between two callback notifications, in seconds.

network→set_callbackUrl(newval)

Changes the callback URL to notify significant state changes.

network→set_discoverable(newval)

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→set_logicalName(newval)

Changes the logical name of the network interface.

network→set_primaryDNS(newval)

Changes the IP address of the primary name server to be used by the module.

network→set_secondaryDNS(newval)

Changes the IP address of the secondary name server to be used by the module.

network→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

network→set_userPassword(newval)

Changes the password for the "user" user.

network→set_wwwWatchdogDelay(newval)

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

network→useStaticIP(ipAddress, subnetMaskLen, router)

Changes the configuration of the network interface to use a static IP address.

network→wait_async(callback, context)

3. Reference

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YNetwork.FindNetwork() yFindNetwork()yFindNetwork()

YNetwork

Retrieves a network interface for a given identifier.

```
YNetwork* yFindNetwork( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the network interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YNetwork.isOnline()` to test if the network interface is indeed online at a given time. In case of ambiguity when looking for a network interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the network interface

Returns :

a `YNetwork` object allowing you to drive the network interface.

YNetwork.FirstNetwork()

YNetwork

yFirstNetwork()yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

YNetwork* **yFirstNetwork()**

Use the method `YNetwork.nextNetwork()` to iterate on next network interfaces.

Returns :

a pointer to a `YNetwork` object, corresponding to the first network interface currently online, or a `null` pointer if there are none.

network→callbackLogin()**network→**
callbackLogin()

YNetwork

Connects to the notification callback and saves the credentials required to log into it.

```
int callbackLogin( string username, string password)
```

The password is not stored into the module, only a hashed copy of the credentials are saved. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

username username required to log to the callback

password password required to log to the callback

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→describe()**network→describe()****YNetwork**

Returns a short text that describes unambiguously the instance of the network interface in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the network interface (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

network→**get_adminPassword()****YNetwork****network**→**adminPassword()****network**→**get_adminPassword()**

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

`string` **get_adminPassword()**

Returns :

a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns `Y_ADMINPASSWORD_INVALID`.

network→**get_advertisedValue()**

YNetwork

network→**advertisedValue()****network**→
get_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the network interface (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

network→get_callbackCredentials()**YNetwork****network→callbackCredentials()****network→****get_callbackCredentials()**

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

string **get_callbackCredentials()**

Returns :

a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns Y_CALLBACKCREDENTIALS_INVALID.

network→get_callbackEncoding()**YNetwork****network→callbackEncoding()****network→****get_callbackEncoding()**

Returns the encoding standard to use for representing notification values.

[Y_CALLBACKENCODING_enum](#) **get_callbackEncoding()**

Returns :

a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV and Y_CALLBACKENCODING_YOCTO_API corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns Y_CALLBACKENCODING_INVALID.

network→get_callbackMaxDelay()**YNetwork****network→callbackMaxDelay()****network→****get_callbackMaxDelay()**

Returns the maximum waiting time between two callback notifications, in seconds.

int **get_callbackMaxDelay()**

Returns :

an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMAXDELAY_INVALID.

network→**get_callbackMethod()****YNetwork****network**→**callbackMethod()****network**→**get_callbackMethod()**

Returns the HTTP method used to notify callbacks for significant state changes.

[Y_CALLBACKMETHOD_enum](#) **get_callbackMethod()**

Returns :

a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns `Y_CALLBACKMETHOD_INVALID`.

network→get_callbackMinDelay()**YNetwork****network→callbackMinDelay()****network→**
get_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

int **get_callbackMinDelay()**

Returns :

an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMINDELAY_INVALID.

network→get_callbackUrl()

YNetwork

network→callbackUrl()

get_callbackUrl()

Returns the callback URL to notify of significant state changes.

`string get_callbackUrl()`

Returns :

a string corresponding to the callback URL to notify of significant state changes

On failure, throws an exception or returns Y_CALLBACKURL_INVALID.

network→get_discoverable()**YNetwork****network→discoverable()****network→**
get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

[Y_DISCOVERABLE_enum](#) **get_discoverable()**

Returns :

either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns Y_DISCOVERABLE_INVALID.

network→**get_errorMessage()**

YNetwork

network→**errorMessage()****network**→

get_errorMessage()

Returns the error message of the latest error with the network interface.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the network interface object

network→**get_errorType()****YNetwork****network**→**errorType()****network**→**get_errorType()**

Returns the numerical error code of the latest error with the network interface.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the network interface object

network→get_friendlyName()**YNetwork****network→friendlyName()**
network→get_friendlyName()

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

string get_friendlyName()

The returned string uses the logical names of the module and of the network interface if they are defined, otherwise the serial number of the module and the hardware identifier of the network interface (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the network interface using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

network→get_functionDescriptor()**YNetwork****network→functionDescriptor()**
network→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

network→get_functionId()

YNetwork

network→functionId()~~network→get_functionId()~~

Returns the hardware identifier of the network interface, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the network interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

network→**get_hardwareId()****YNetwork****network**→**hardwareId()****network**→
get_hardwareId()

Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the network interface. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the network interface (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

network→get_ipAddress()

YNetwork

network→ipAddress()~~network→get_ipAddress()~~

Returns the IP address currently in use by the device.

string **get_ipAddress()**

The address may have been configured statically, or provided by a DHCP server.

Returns :

a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns Y_IPADDRESS_INVALID.

network→**get_logicalName()**
network→**logicalName()****network**→
get_logicalName()

YNetwork

Returns the logical name of the network interface.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the network interface. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

network→get_macAddress()

YNetwork

network→macAddress()**network→**

get_macAddress()

Returns the MAC address of the network interface.

string get_macAddress()

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

Returns :

a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns Y_MACADDRESS_INVALID.

network→get_module()**YNetwork****network→module()**~~network→get_module()~~

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

network→**get_poeCurrent()****YNetwork****network**→**poeCurrent()****network**→**get_poeCurrent ()**

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

int **get_poeCurrent()**

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

Returns :

an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns Y_POECURRENT_INVALID.

network→**get_primaryDNS()**
network→**primaryDNS()****network**→
get_primaryDNS()

YNetwork

Returns the IP address of the primary name server to be used by the module.

string **get_primaryDNS()**

Returns :

a string corresponding to the IP address of the primary name server to be used by the module

On failure, throws an exception or returns Y_PRIMARYDNS_INVALID.

network→get_readiness()**YNetwork****network→readiness()****network→get_readiness()**

Returns the current established working mode of the network interface.

`Y_READINESS_enum get_readiness()`

Level zero (DOWN_0) means that no hardware link has been detected. Either there is no signal on the network cable, or the selected wireless access point cannot be detected. Level 1 (LIVE_1) is reached when the network is detected, but is not yet connected. For a wireless network, this shows that the requested SSID is present. Level 2 (LINK_2) is reached when the hardware connection is established. For a wired network connection, level 2 means that the cable is attached at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS_4) is reached when the DNS server is reachable on the network. Level 5 (WWW_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

Returns :

a value among Y_READINESS_DOWN, Y_READINESS_EXISTS, Y_READINESS_LINKED, Y_READINESS_LAN_OK and Y_READINESS_WWW_OK corresponding to the current established working mode of the network interface

On failure, throws an exception or returns Y_READINESS_INVALID.

network→get_router()**YNetwork****network→router()****network→get_router()**

Returns the IP address of the router on the device subnet (default gateway).

string **get_router()**

Returns :

a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns Y_ROUTER_INVALID.

network→**get_secondaryDNS()**

YNetwork

network→**secondaryDNS()****network**→

get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

string **get_secondaryDNS()**

Returns :

a string corresponding to the IP address of the secondary name server to be used by the module

On failure, throws an exception or returns `Y_SECONDARYDNS_INVALID`.

network→**get_subnetMask()**
network→**subnetMask()****network**→
get_subnetMask()

YNetwork

Returns the subnet mask currently used by the device.

string **get_subnetMask()**

Returns :

a string corresponding to the subnet mask currently used by the device

On failure, throws an exception or returns Y_SUBNETMASK_INVALID.

network→get_userdata()

YNetwork

network→userdata()~~network→get_userdata()~~

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

network→**get_userPassword()**
network→**userPassword()****network**→
get_userPassword()

YNetwork

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

string **get_userPassword()**

Returns :

a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns Y_USERPASSWORD_INVALID.

network→**get_wwwWatchdogDelay()****YNetwork****network**→**wwwWatchdogDelay()****network**→**get_wwwWatchdogDelay()**

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

int **get_wwwWatchdogDelay()**

A zero value disables automated reboot in case of Internet connectivity loss.

Returns :

an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

On failure, throws an exception or returns `Y_WWWWATCHDOGDELAY_INVALID`.

network→isOnline()**network→isOnline()****YNetwork**

Checks if the network interface is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

Returns :

`true` if the network interface can be reached, and `false` otherwise

network→load()**network→load()****YNetwork**

Preloads the network interface cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

network→nextNetwork()**network→nextNetwork()****YNetwork**

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

`YNetwork * nextNetwork()`

Returns :

a pointer to a `YNetwork` object, corresponding to a network interface currently online, or a `null` pointer if there are no more network interfaces to enumerate.

network→ping()**network→ping()****YNetwork**

Pings str_host to test the network connectivity.

```
string ping( string host)
```

Sends four ICMP ECHO_REQUEST requests from the module to the target str_host. This method returns a string with the result of the 4 ICMP ECHO_REQUEST requests.

Parameters :

host the hostname or the IP address of the target

Returns :

a string with the result of the ping.

network→registerValueCallback()network→
registerValueCallback()

YNetwork

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YNetworkValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

network→**set_adminPassword()**
network→**setAdminPassword()****network**→
set_adminPassword()

Changes the password for the "admin" user.

```
int set_adminPassword( const string& newval)
```

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "admin" user

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackCredentials()**YNetwork****network→setCallbackCredentials()**
network→set_callbackCredentials()

Changes the credentials required to connect to the callback address.

```
int set_callbackCredentials( const string& newval)
```

The credentials must be provided as returned by function `get_callbackCredentials`, in the form `username:hash`. The method used to compute the hash varies according to the authentication scheme implemented by the callback, For Basic authentication, the hash is the MD5 of the string `username:password`. For Digest authentication, the hash is the MD5 of the string `username:realm:password`. For a simpler way to configure callback credentials, use function `callbackLogin` instead. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the credentials required to connect to the callback address

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackEncoding()****YNetwork****network**→**setCallbackEncoding()****network**→
set_callbackEncoding()

Changes the encoding standard to use for representing notification values.

```
int set_callbackEncoding( Y_CALLBACKENCODING_enum newval)
```

Parameters :

newval a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV and Y_CALLBACKENCODING_YOCTO_API corresponding to the encoding standard to use for representing notification values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMaxDelay()**YNetwork****network→setCallbackMaxDelay()**
network→set_callbackMaxDelay()

Changes the maximum waiting time between two callback notifications, in seconds.

```
int set_callbackMaxDelay( int newval)
```

Parameters :

newval an integer corresponding to the maximum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackMethod()****YNetwork****network**→**setCallbackMethod()****network**→**set_callbackMethod()**

Changes the HTTP method used to notify callbacks for significant state changes.

```
int set_callbackMethod( Y_CALLBACKMETHOD_enum newval)
```

Parameters :

newval a value among Y_CALLBACKMETHOD_POST, Y_CALLBACKMETHOD_GET and Y_CALLBACKMETHOD_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMinDelay()**YNetwork****network→setCallbackMinDelay()**
network→set_callbackMinDelay()

Changes the minimum waiting time between two callback notifications, in seconds.

```
int set_callbackMinDelay( int newval)
```

Parameters :

newval an integer corresponding to the minimum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackUrl()****YNetwork****network**→**setCallbackUrl()****network**→**set_callbackUrl()**

Changes the callback URL to notify significant state changes.

```
int set_callbackUrl( const string& newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the callback URL to notify significant state changes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_discoverable()****YNetwork****network**→**setDiscoverable()****network**→
set_discoverable()

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
int set_discoverable( Y_DISCOVERABLE_enum newval)
```

Parameters :

newval either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_logicalName()****YNetwork****network**→**setLogicalName()****network**→**set_logicalName()**

Changes the logical name of the network interface.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the network interface.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

network→set_primaryDNS()**YNetwork****network→setPrimaryDNS()**
network→set_primaryDNS()

Changes the IP address of the primary name server to be used by the module.

```
int set_primaryDNS( const string& newval)
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the primary name server to be used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_secondaryDNS()****YNetwork****network**→**setSecondaryDNS()****network**→**set_secondaryDNS ()**

Changes the IP address of the secondary name server to be used by the module.

```
int set_secondaryDNS( const string& newval)
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the secondary name server to be used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_userdata()**YNetwork****network→setUserData()****network→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

network→**set_userPassword()**
network→**setUserPassword()****network**→
set_userPassword()

Changes the password for the "user" user.

```
int set_userPassword( const string& newval)
```

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "user" user

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_wwwWatchdogDelay()****YNetwork****network**→**setWwwWatchdogDelay()****network**→**set_wwwWatchdogDelay()**

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
int set_wwwWatchdogDelay( int newval)
```

A zero value disables automated reboot in case of Internet connectivity loss. The smallest valid non-zero timeout is 90 seconds.

Parameters :

newval an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useDHCP()**network→useDHCP ()****YNetwork**

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

```
int useDHCP( string fallbackIpAddr,  
             int fallbackSubnetMaskLen,  
             string fallbackRouter)
```

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

fallbackIpAddr	fallback IP address, to be used when no DHCP reply is received
fallbackSubnetMaskLen	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
fallbackRouter	fallback router IP address, to be used when no DHCP reply is received

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useStaticIP()

YNetwork

Changes the configuration of the network interface to use a static IP address.

```
int useStaticIP( string ipAddress,
                int subnetMaskLen,
                string router)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ipAddress device IP address
subnetMaskLen subnet mask length, as an integer (eg. 24 means 255.255.255.0)
router router IP address (default gateway)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.27. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_oscontrol.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YOsControl = yoctolib.YOsControl;
php	require_once('yocto_oscontrol.php');
c++	#include "yocto_oscontrol.h"
m	#import "yocto_oscontrol.h"
pas	uses yocto_oscontrol;
vb	yocto_oscontrol.vb
cs	yocto_oscontrol.cs
java	import com.yoctopuce.YoctoAPI.YOsControl;
py	from yocto_oscontrol import *

Global functions

yFindOsControl(func)

Retrieves OS control for a given identifier.

yFirstOsControl()

Starts the enumeration of OS control currently accessible.

YOsControl methods

oscontrol→describe()

Returns a short text that describes unambiguously the instance of the OS control in the form TYPE (NAME) = SERIAL . FUNCTIONID.

oscontrol→get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

oscontrol→get_errorMessage()

Returns the error message of the latest error with the OS control.

oscontrol→get_errorType()

Returns the numerical error code of the latest error with the OS control.

oscontrol→get_friendlyName()

Returns a global identifier of the OS control in the format MODULE_NAME . FUNCTION_NAME.

oscontrol→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

oscontrol→get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

oscontrol→get_hardwareId()

Returns the unique hardware identifier of the OS control in the form SERIAL . FUNCTIONID.

oscontrol→get_logicalName()

Returns the logical name of the OS control.

oscontrol→get_module()

Gets the YModule object for the device on which the function is located.

oscontrol→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`oscontrol→get_shutdownCountdown()`

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

`oscontrol→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

`oscontrol→isOnline()`

Checks if the OS control is currently reachable, without raising any error.

`oscontrol→isOnline_async(callback, context)`

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

`oscontrol→load(msValidity)`

Preloads the OS control cache with a specified validity duration.

`oscontrol→load_async(msValidity, callback, context)`

Preloads the OS control cache with a specified validity duration (asynchronous version).

`oscontrol→nextOsControl()`

Continues the enumeration of OS control started using `yFirstOsControl()`.

`oscontrol→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`oscontrol→set_logicalName(newval)`

Changes the logical name of the OS control.

`oscontrol→set_userData(data)`

Stores a user context provided as argument in the `userData` attribute of the function.

`oscontrol→shutdown(secBeforeShutDown)`

Schedules an OS shutdown after a given number of seconds.

`oscontrol→wait_async(callback, context)`

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YOsControl.FindOsControl()**YOsControl****yFindOsControl()****yFindOsControl()**

Retrieves OS control for a given identifier.

```
YOsControl* yFindOsControl( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YOsControl.isOnline()` to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the OS control

Returns :

a `YOsControl` object allowing you to drive the OS control.

YOsControl.FirstOsControl()
yFirstOsControl()**yFirstOsControl()****YOsControl**

Starts the enumeration of OS control currently accessible.

`YOsControl* yFirstOsControl()`

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

Returns :

a pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a `null` pointer if there are none.

oscontrol→describe()**oscontrol→describe()****YOsControl**

Returns a short text that describes unambiguously the instance of the OS control in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the OS control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

oscontrol→**get_advertisedValue()****YOsControl****oscontrol**→**advertisedValue()****oscontrol**→
get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the OS control (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

oscontrol→get_errorMessage()

YOsControl

oscontrol→errorMessage()
oscontrol→
get_errorMessage()

Returns the error message of the latest error with the OS control.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the OS control object

oscontrol→**get_errorType()****YOsControl****oscontrol**→**errorType()****oscontrol**→
get_errorType()

Returns the numerical error code of the latest error with the OS control.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the OS control object

oscontrol→**get_friendlyName()**

YOsControl

oscontrol→**friendlyName()****oscontrol**→

get_friendlyName()

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the OS control if they are defined, otherwise the serial number of the module and the hardware identifier of the OS control (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the OS control using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

oscontrol→get_functionDescriptor()**YOsControl****oscontrol→functionDescriptor()oscontrol→
get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

oscontrol→**get_functionId()**

YOsControl

oscontrol→**functionId()****oscontrol**→

get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

string **get_functionId()**

For example relay1

Returns :

a string that identifies the OS control (ex: relay1) On failure, throws an exception or returns Y_FUNCTIONID_INVALID.

oscontrol→**get_hardwareId()****YOsControl****oscontrol**→**hardwareId()****oscontrol**→
get_hardwareId()

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the OS control. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the OS control (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

oscontrol→**get_logicalName()**

YOsControl

oscontrol→**logicalName()****oscontrol**→

get_logicalName()

Returns the logical name of the OS control.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the OS control. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

oscontrol→get_module()**YOsControl****oscontrol→module()****oscontrol→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

oscontrol→get_shutdownCountdown()

YOsControl

oscontrol→shutdownCountdown()oscontrol→

get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

int get_shutdownCountdown()

Returns :

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns Y_SHUTDOWNCOUNTDOWN_INVALID.

oscontrol→get_userdata()**YOsControl****oscontrol→userdata()****oscontrol→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

oscontrol→isOnline()**oscontrol→isOnline()**

YOsControl

Checks if the OS control is currently reachable, without raising any error.

`bool isOnline()`

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

Returns :

`true` if the OS control can be reached, and `false` otherwise

oscontrol→load()**oscontrol→load()****YOsControl**

Preloads the OS control cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

oscontrol→**nextOsControl()**oscontrol→
nextOsControl ()

YOsControl

Continues the enumeration of OS control started using `yFirstOsControl ()`.

`YOsControl * nextOsControl()`

Returns :

a pointer to a `YOsControl` object, corresponding to OS control currently online, or a `null` pointer if there are no more OS control to enumerate.

oscontrol→registerValueCallback()**oscontrol→**
registerValueCallback()

YOsControl

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YOsControlValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

oscontrol→**set_logicalName()****YOsControl****oscontrol**→**setLogicalName()****oscontrol**→**set_logicalName()**

Changes the logical name of the OS control.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the OS control.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

oscontrol→set_userdata()**YOsControl****oscontrol→setUserData()**
oscontrol→
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

oscontrol→shutdown()**oscontrol→shutdown()****YOsControl**

Schedules an OS shutdown after a given number of seconds.

```
int shutdown( int secBeforeShutDown)
```

Parameters :

secBeforeShutDown number of seconds before shutdown

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

3.28. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_power.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPower = yoctolib.YPower;
php	require_once('yocto_power.php');
c++	#include "yocto_power.h"
m	#import "yocto_power.h"
pas	uses yocto_power;
vb	yocto_power.vb
cs	yocto_power.cs
java	import com.yoctopuce.YoctoAPI.YPower;
py	from yocto_power import *

Global functions

yFindPower(func)

Retrieves a electrical power sensor for a given identifier.

yFirstPower()

Starts the enumeration of electrical power sensors currently accessible.

YPower methods

power→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

power→describe()

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

power→get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

power→get_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

power→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

power→get_currentValue()

Returns the current measure for the electrical power.

power→get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

power→get_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

power→get_friendlyName()

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME . FUNCTION_NAME`.

power→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

power→**get_hardwareId()**

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

power→**get_highestValue()**

Returns the maximal value observed for the electrical power.

power→**get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→**get_logicalName()**

Returns the logical name of the electrical power sensor.

power→**get_lowestValue()**

Returns the minimal value observed for the electrical power.

power→**get_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

power→**get_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

power→**get_module()**

Gets the `YModule` object for the device on which the function is located.

power→**get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

power→**get_recordedData(startTime, endTime)**

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

power→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→**get_resolution()**

Returns the resolution of the measured values.

power→**get_unit()**

Returns the measuring unit for the electrical power.

power→**get_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

power→**isOnline()**

Checks if the electrical power sensor is currently reachable, without raising any error.

power→**isOnline_async(callback, context)**

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

power→**load(msValidity)**

Preloads the electrical power sensor cache with a specified validity duration.

power→**loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

power→**load_async(msValidity, callback, context)**

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

power→**nextPower()**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

power→**registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

power→**registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

power→reset()

Resets the energy counter.

power→set_highestValue(newval)

Changes the recorded maximal value observed pour the electrical power.

power→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

power→set_logicalName(newval)

Changes the logical name of the electrical power sensor.

power→set_lowestValue(newval)

Changes the recorded minimal value observed pour the electrical power.

power→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

power→set_resolution(newval)

Changes the resolution of the measured values.

power→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

power→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPower.FindPower()**YPower****yFindPower()****yFindPower()**

Retrieves a electrical power sensor for a given identifier.

```
YPower* yFindPower( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the electrical power sensor

Returns :

a `YPower` object allowing you to drive the electrical power sensor.

YPower.FirstPower()
yFirstPower()**yFirstPower()**

YPower

Starts the enumeration of electrical power sensors currently accessible.

`YPower* yFirstPower()`

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

Returns :

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

power→calibrateFromPoints()power→
calibrateFromPoints()

YPower

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→describe()**power→describe()****YPower**

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the electrical power sensor (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

power→**get_advertisedValue()****YPower****power**→**advertisedValue()****power**→
get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

string **get_advertisedValue()****Returns :**

a string corresponding to the current value of the electrical power sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

power→get_cosPhi()**YPower****power→cosPhi()****power→get_cosPhi()**

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

double **get_cosPhi()****Returns :**

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns Y_COSPHI_INVALID.

power→**get_currentRawValue()**

YPower

power→**currentRawValue()****power**→

get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

power→**get_currentValue()****YPower****power**→**currentValue()****power**→
get_currentValue()

Returns the current measure for the electrical power.

`double` **get_currentValue()**

Returns :

a floating point number corresponding to the current measure for the electrical power

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

power→get_errorMessage()

YPower

power→errorMessage()
power→
get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the electrical power sensor object

power→**get_errorType()****YPower****power**→**errorType()****power**→**get_errorType()**

Returns the numerical error code of the latest error with the electrical power sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the electrical power sensor object

power→**get_friendlyName()****YPower****power**→**friendlyName()****power**→
get_friendlyName()

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the electrical power sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

power→**get_functionDescriptor()**
power→**functionDescriptor()****power**→
get_functionDescriptor()

YPower

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

power→**get_functionId()**

YPower

power→**functionId()****power**→**get_functionId()**

Returns the hardware identifier of the electrical power sensor, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the electrical power sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

power→get_hardwareId()**YPower****power→hardwareId()****power→get_hardwareId()**

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the electrical power sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the electrical power sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

power→**get_highestValue()**

YPower

power→**highestValue()****power**→

get_highestValue()

Returns the maximal value observed for the electrical power.

`double get_highestValue()`

Returns :

a floating point number corresponding to the maximal value observed for the electrical power

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

power→get_logFrequency()**YPower****power→logFrequency()****power→**
get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

power→**get_logicalName()**

YPower

power→**logicalName()****power**→**get_logicalName()**

Returns the logical name of the electrical power sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the electrical power sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

power→**get_lowestValue()****YPower****power**→**lowestValue()****power**→**get_lowestValue()**

Returns the minimal value observed for the electrical power.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the electrical power

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

power→**get_meter()**

YPower

power→**meter()****power**→**get_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

double **get_meter()**

Note that this counter is reset at each start of the device.

Returns :

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns Y_METER_INVALID.

power→get_meterTimer()**YPower****power→meterTimer()****power→get_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

int **get_meterTimer()**

Returns :

an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns Y_METERTIMER_INVALID.

power→get_module()

YPower

power→module()**power→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

power→**get_recordedData()****YPower****power**→**recordedData()****power**→**get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YDataSet get_recordedData( s64 startTime, s64 endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

power→**get_reportFrequency()****YPower****power**→**reportFrequency()****power**→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()****Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

power→**get_resolution()****YPower****power**→**resolution()****power**→**get_resolution()**

Returns the resolution of the measured values.

double **get_resolution()** ()

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

power→**get_unit()**

YPower

power→**unit()****power**→**get_unit()**

Returns the measuring unit for the electrical power.

string **get_unit()**

Returns :

a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns Y_UNIT_INVALID.

power→get_userdata()**YPower****power→userdata()****power→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

power→isOnline()**power→isOnline()**

YPower

Checks if the electrical power sensor is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

Returns :

`true` if the electrical power sensor can be reached, and `false` otherwise

power→load()**power→load()****YPower**

Preloads the electrical power sensor cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

power→**loadCalibrationPoints()****power**→
loadCalibrationPoints()

YPower

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→nextPower()**power→nextPower ()****YPower**

Continues the enumeration of electrical power sensors started using `yFirstPower ()`.

`YPower * nextPower()`

Returns :

a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a `null` pointer if there are no more electrical power sensors to enumerate.

power→**registerTimedReportCallback()****power**→
registerTimedReportCallback()

YPower

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YPowerTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

power→**registerValueCallback()****power**→
registerValueCallback()

YPower

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YPowerValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

power→reset()**power→reset ()**

YPower

Resets the energy counter.

int reset()

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_highestValue()****YPower****power**→**setHighestValue()****power**→
set_highestValue()

Changes the recorded maximal value observed pour the electrical power.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed pour the electrical power

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_logFrequency()****YPower****power**→**setLogFrequency()****power**→
set_logFrequency()

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_logicalName()****YPower****power**→**setLogicalName()****power**→
set_logicalName()

Changes the logical name of the electrical power sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the electrical power sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

power→**set_lowestValue()****YPower****power**→**setLowestValue()****power**→
set_lowestValue()

Changes the recorded minimal value observed pour the electrical power.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed pour the electrical power

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_reportFrequency()****YPower****power**→**setReportFrequency()****power**→**set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_resolution()****YPower****power**→**setResolution()****power**→**set_resolution()**

Changes the resolution of the measured values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_userdata()**YPower****power→setUserData()**~~power→set_userdata()~~

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.29. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pressure.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPressure = yoctolib.YPressure;
php	require_once('yocto_pressure.php');
c++	#include "yocto_pressure.h"
m	#import "yocto_pressure.h"
pas	uses yocto_pressure;
vb	yocto_pressure.vb
cs	yocto_pressure.cs
java	import com.yoctopuce.YoctoAPI.YPressure;
py	from yocto_pressure import *

Global functions

yFindPressure(func)

Retrieves a pressure sensor for a given identifier.

yFirstPressure()

Starts the enumeration of pressure sensors currently accessible.

YPressure methods

pressure→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

pressure→describe()

Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

pressure→get_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

pressure→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

pressure→get_currentValue()

Returns the current measure for the pressure.

pressure→get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

pressure→get_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

pressure→get_friendlyName()

Returns a global identifier of the pressure sensor in the format `MODULE_NAME . FUNCTION_NAME`.

pressure→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pressure→get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

pressure→get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL . FUNCTIONID`.

pressure→get_highestValue()

Returns the maximal value observed for the pressure.

pressure→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pressure→get_logicalName()

Returns the logical name of the pressure sensor.

pressure→get_lowestValue()

Returns the minimal value observed for the pressure.

pressure→get_module()

Gets the YModule object for the device on which the function is located.

pressure→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

pressure→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pressure→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pressure→get_resolution()

Returns the resolution of the measured values.

pressure→get_unit()

Returns the measuring unit for the pressure.

pressure→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

pressure→isOnline()

Checks if the pressure sensor is currently reachable, without raising any error.

pressure→isOnline_async(callback, context)

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

pressure→load(msValidity)

Preloads the pressure sensor cache with a specified validity duration.

pressure→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

pressure→load_async(msValidity, callback, context)

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

pressure→nextPressure()

Continues the enumeration of pressure sensors started using yFirstPressure().

pressure→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

pressure→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pressure→set_highestValue(newval)

Changes the recorded maximal value observed for the pressure.

pressure→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

pressure→set_logicalName(newval)

Changes the logical name of the pressure sensor.

3. Reference

pressure→**set_lowestValue**(newval)

Changes the recorded minimal value observed for the pressure.

pressure→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

pressure→**set_resolution**(newval)

Changes the resolution of the measured physical values.

pressure→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

pressure→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPressure.FindPressure() yFindPressure()yFindPressure()

YPressure

Retrieves a pressure sensor for a given identifier.

```
YPressure* yFindPressure( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the pressure sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPressure.isOnline()` to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the pressure sensor

Returns :

a `YPressure` object allowing you to drive the pressure sensor.

YPressure.FirstPressure()

YPressure

yFirstPressure()`yFirstPressure()`

Starts the enumeration of pressure sensors currently accessible.

`YPressure* yFirstPressure()`

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()****pressure**→
calibrateFromPoints()

YPressure

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→describe()**pressure→describe()****YPressure**

Returns a short text that describes unambiguously the instance of the pressure sensor in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the pressure sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pressure→**get_advertisedValue()****YPressure****pressure**→**advertisedValue()****pressure**→**get_advertisedValue()**

Returns the current value of the pressure sensor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the pressure sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pressure→**get_currentRawValue()**

YPressure

pressure→**currentRawValue()****pressure**→

get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

pressure→**get_currentValue()****YPressure****pressure**→**currentValue()****pressure**→
get_currentValue()

Returns the current measure for the pressure.

`double` **get_currentValue()**

Returns :

a floating point number corresponding to the current measure for the pressure

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

pressure→**get_errorMessage()**

YPressure

pressure→**errorMessage()****pressure**→

get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the pressure sensor object

pressure→**get_errorType()****YPressure****pressure**→**errorType()****pressure**→**get_errorType()**

Returns the numerical error code of the latest error with the pressure sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the pressure sensor object

pressure→**get_friendlyName()****YPressure****pressure**→**friendlyName()****pressure**→
get_friendlyName()

Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the pressure sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the pressure sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the pressure sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pressure→**get_functionDescriptor()****YPressure****pressure**→**functionDescriptor()****pressure**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pressure→**get_functionId()**

YPressure

pressure→**functionId()****pressure**→
get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the pressure sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pressure→**get_hardwareId()****YPressure****pressure**→**hardwareId()****pressure**→
get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()** ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the pressure sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the pressure sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

pressure→**get_highestValue()**

YPressure

pressure→**highestValue()****pressure**→

get_highestValue()

Returns the maximal value observed for the pressure.

`double get_highestValue()`

Returns :

a floating point number corresponding to the maximal value observed for the pressure

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

pressure→get_logFrequency()**YPressure****pressure→logFrequency()****pressure→****get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
string get_logFrequency( )
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

pressure→**get_logicalName()**

YPressure

pressure→**logicalName()****pressure**→
get_logicalName()

Returns the logical name of the pressure sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the pressure sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pressure→**get_lowestValue()****YPressure****pressure**→**lowestValue()****pressure**→
get_lowestValue()

Returns the minimal value observed for the pressure.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the pressure

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

pressure→get_module()

YPressure

pressure→module()**pressure→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

pressure→**get_recordedData()****YPressure****pressure**→**recordedData()****pressure**→**get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YDataSet get_recordedData( s64 startTime, s64 endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pressure→**get_reportFrequency()**

YPressure

pressure→**reportFrequency()****pressure**→

get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

pressure→**get_resolution()****YPressure****pressure**→**resolution()****pressure**→
get_resolution()

Returns the resolution of the measured values.

`double` **get_resolution()**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

pressure→**get_unit()**

YPressure

pressure→**unit()****pressure**→**get_unit()**

Returns the measuring unit for the pressure.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns Y_UNIT_INVALID.

pressure→get_userdata()**YPressure****pressure→userData()****pressure→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

pressure→**isOnline()****pressure**→**isOnline()**

YPressure

Checks if the pressure sensor is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

Returns :

`true` if the pressure sensor can be reached, and `false` otherwise

pressure→load()**pressure→load()****YPressure**

Preloads the pressure sensor cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pressure→**loadCalibrationPoints()****pressure**→
loadCalibrationPoints()

YPressure

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**nextPressure()****pressure**→
nextPressure()

YPressure

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

`YPressure * nextPressure()`

Returns :

a pointer to a `YPressure` object, corresponding to a pressure sensor currently online, or a `null` pointer if there are no more pressure sensors to enumerate.

pressure → registerTimedReportCallback() pressure	YPressure
→ registerTimedReportCallback()	

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YPressureTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

pressure→**registerValueCallback()****pressure**→
registerValueCallback()

YPressure

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YPressureValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pressure→**set_highestValue()****YPressure****pressure**→**setHighestValue()****pressure**→
set_highestValue()

Changes the recorded maximal value observed for the pressure.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the pressure

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logFrequency()**YPressure****pressure→setLogFrequency()**
pressure→set_logFrequency()

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**set_logicalName()****YPressure****pressure**→**setLogicalName()****pressure**→**set_logicalName()**

Changes the logical name of the pressure sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the pressure sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

pressure→**set_lowestValue()****YPressure****pressure**→**setLowestValue()****pressure**→
set_lowestValue()

Changes the recorded minimal value observed for the pressure.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the pressure

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_reportFrequency()**YPressure****pressure→setReportFrequency()**
pressure→set_reportFrequency()

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**set_resolution()****YPressure****pressure**→**setResolution()****pressure**→
set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_userdata()

YPressure

pressure→setUserData()**pressure→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.30. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmoutput.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPwmOutput = yoctolib.YPwmOutput;
php	require_once('yocto_pwmoutput.php');
c++	#include "yocto_pwmoutput.h"
m	#import "yocto_pwmoutput.h"
pas	uses yocto_pwmoutput;
vb	yocto_pwmoutput.vb
cs	yocto_pwmoutput.cs
java	import com.yoctopuce.YoctoAPI.YPwmOutput;
py	from yocto_pwmoutput import *

Global functions

yFindPwmOutput(func)

Retrieves a PWM for a given identifier.

yFirstPwmOutput()

Starts the enumeration of PWMs currently accessible.

YPwmOutput methods

pwmoutput→describe()

Returns a short text that describes unambiguously the instance of the PWM in the form TYPE (NAME) = SERIAL . FUNCTIONID.

pwmoutput→dutyCycleMove(target, ms_duration)

Performs a smooth change of the pulse duration toward a given value.

pwmoutput→get_advertisedValue()

Returns the current value of the PWM (no more than 6 characters).

pwmoutput→get_dutyCycle()

Returns the PWM duty cycle, in per cents.

pwmoutput→get_dutyCycleAtPowerOn()

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

pwmoutput→get_enabled()

Returns the state of the PWMs.

pwmoutput→get_enabledAtPowerOn()

Returns the state of the PWM at device power on.

pwmoutput→get_errorMessage()

Returns the error message of the latest error with the PWM.

pwmoutput→get_errorType()

Returns the numerical error code of the latest error with the PWM.

pwmoutput→get_frequency()

Returns the PWM frequency in Hz.

pwmoutput→get_friendlyName()

Returns a global identifier of the PWM in the format MODULE_NAME . FUNCTION_NAME.

pwmoutput→get_functionDescriptor()

	Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
<code>pwmoutput→get_functionId()</code>	Returns the hardware identifier of the PWM, without reference to the module.
<code>pwmoutput→get_hardwareId()</code>	Returns the unique hardware identifier of the PWM in the form <code>SERIAL . FUNCTIONID</code> .
<code>pwmoutput→get_logicalName()</code>	Returns the logical name of the PWM.
<code>pwmoutput→get_module()</code>	Gets the <code>YModule</code> object for the device on which the function is located.
<code>pwmoutput→get_module_async(callback, context)</code>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<code>pwmoutput→get_period()</code>	Returns the PWM period in milliseconds.
<code>pwmoutput→get_pulseDuration()</code>	Returns the PWM pulse length in milliseconds.
<code>pwmoutput→get_userData()</code>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<code>pwmoutput→isOnline()</code>	Checks if the PWM is currently reachable, without raising any error.
<code>pwmoutput→isOnline_async(callback, context)</code>	Checks if the PWM is currently reachable, without raising any error (asynchronous version).
<code>pwmoutput→load(msValidity)</code>	Preloads the PWM cache with a specified validity duration.
<code>pwmoutput→load_async(msValidity, callback, context)</code>	Preloads the PWM cache with a specified validity duration (asynchronous version).
<code>pwmoutput→nextPwmOutput()</code>	Continues the enumeration of PWMs started using <code>yFirstPwmOutput()</code> .
<code>pwmoutput→pulseDurationMove(ms_target, ms_duration)</code>	Performs a smooth transision of the pulse duration toward a given value.
<code>pwmoutput→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>pwmoutput→set_dutyCycle(newval)</code>	Changes the PWM duty cycle, in per cents.
<code>pwmoutput→set_dutyCycleAtPowerOn(newval)</code>	Changes the PWM duty cycle at device power on.
<code>pwmoutput→set_enabled(newval)</code>	Stops or starts the PWM.
<code>pwmoutput→set_enabledAtPowerOn(newval)</code>	Changes the state of the PWM at device power on.
<code>pwmoutput→set_frequency(newval)</code>	Changes the PWM frequency.
<code>pwmoutput→set_logicalName(newval)</code>	Changes the logical name of the PWM.
<code>pwmoutput→set_period(newval)</code>	Changes the PWM period.

pwmoutput→set_pulseDuration(newval)

Changes the PWM pulse length, in milliseconds.

pwmoutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmoutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmOutput.FindPwmOutput()**YPwmOutput****yFindPwmOutput()****yFindPwmOutput()**

Retrieves a PWM for a given identifier.

```
YPwmOutput* yFindPwmOutput( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the PWM

Returns :

a `YPwmOutput` object allowing you to drive the PWM.

YPwmOutput.FirstPwmOutput()**YPwmOutput****yFirstPwmOutput()****yFirstPwmOutput()**

Starts the enumeration of PWMs currently accessible.

YPwmOutput* yFirstPwmOutput()

Use the method `YPwmOutput.nextPwmOutput()` to iterate on next PWMs.

Returns :

a pointer to a `YPwmOutput` object, corresponding to the first PWM currently online, or a `null` pointer if there are none.

pwmoutput→describe()**YPwmOutput**

Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the PWM (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmoutput→**dutyCycleMove()****pwmoutput**→
dutyCycleMove()

YPwmOutput

Performs a smooth change of the pulse duration toward a given value.

```
int dutyCycleMove( double target, int ms_duration)
```

Parameters :

target new duty cycle at the end of the transition (floating-point number, between 0 and 1)
ms_duration total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→get_advertisedValue()

YPwmOutput

pwmoutput→advertisedValue()
pwmoutput→get_advertisedValue()

Returns the current value of the PWM (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the PWM (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwmoutput→get_dutyCycle()**YPwmOutput****pwmoutput→dutyCycle()**
pwmoutput→get_dutyCycle()

Returns the PWM duty cycle, in per cents.

double **get_dutyCycle()**

Returns :

a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns Y_DUTYCYCLE_INVALID.

pwmoutput→get_dutyCycleAtPowerOn()

YPwmOutput

pwmoutput→dutyCycleAtPowerOn()
pwmoutput→get_dutyCycleAtPowerOn()

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

double get_dutyCycleAtPowerOn()

Returns :

a floating point number corresponding to the PWMs duty cycle at device power on as a floating point number between 0 and 100

On failure, throws an exception or returns Y_DUTYCYCLEATPOWERON_INVALID.

pwmoutput→get_enabled()**YPwmOutput****pwmoutput→enabled()****pwmoutput→get_enabled()**

Returns the state of the PWMs.

`Y_ENABLED_enum` **get_enabled()**

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the state of the PWMs

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

pwmoutput→get_enabledAtPowerOn()

YPwmOutput

pwmoutput→enabledAtPowerOn()
pwmoutput→get_enabledAtPowerOn()

Returns the state of the PWM at device power on.

[Y_ENABLEDATPOWERON_enum](#) **get_enabledAtPowerOn()**

Returns :

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

pwmoutput→get_errorMessage()**YPwmOutput****pwmoutput→errorMessage()**
pwmoutput→
get_errorMessage()

Returns the error message of the latest error with the PWM.

```
string get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the PWM object

pwmoutput→get_errorType()

YPwmOutput

pwmoutput→errorType()
pwmoutput→get_errorType()

Returns the numerical error code of the latest error with the PWM.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the PWM object

pwmoutput→get_frequency()**YPwmOutput****pwmoutput→frequency()****pwmoutput→**
get_frequency()

Returns the PWM frequency in Hz.

int **get_frequency()**

Returns :

an integer corresponding to the PWM frequency in Hz

On failure, throws an exception or returns Y_FREQUENCY_INVALID.

pwmoutput→**get_friendlyName()****YPwmOutput****pwmoutput**→**friendlyName()****pwmoutput**→
get_friendlyName()

Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the PWM if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the PWM using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwmoutput→**get_functionDescriptor()****YPwmOutput****pwmoutput**→**functionDescriptor()****pwmoutput**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmoutput→get_functionId()

YPwmOutput

pwmoutput→functionId()
pwmoutput→get_functionId()

Returns the hardware identifier of the PWM, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the PWM (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwmoutput→get_hardwareId()**YPwmOutput****pwmoutput→hardwareId()**
pwmoutput→get_hardwareId()

Returns the unique hardware identifier of the PWM in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the PWM (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

pwmoutput→get_logicalName()

YPwmOutput

pwmoutput→logicalName()
pwmoutput→get_logicalName()

Returns the logical name of the PWM.

string get_logicalName()

Returns :

a string corresponding to the logical name of the PWM. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pwmoutput→get_module()**YPwmOutput****pwmoutput→module()****pwmoutput→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

pwmoutput→get_period()

YPwmOutput

pwmoutput→period()**pwmoutput→get_period()**

Returns the PWM period in milliseconds.

double **get_period()** ()

Returns :

a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns Y_PERIOD_INVALID.

pwmoutput→get_pulseDuration()**YPwmOutput****pwmoutput→pulseDuration()**
pwmoutput→
get_pulseDuration()

Returns the PWM pulse length in milliseconds.

```
double get_pulseDuration( )
```

Returns :

a floating point number corresponding to the PWM pulse length in milliseconds

On failure, throws an exception or returns Y_PULSEDURATION_INVALID.

pwmoutput→get_userdata()

YPwmOutput

pwmoutput→userdata()
get_userdata()

Returns the value of the `userData` attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

pwmoutput→isOnline()**pwmoutput→isOnline()****YPwmOutput**

Checks if the PWM is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

Returns :

`true` if the PWM can be reached, and `false` otherwise

pwmoutput→load()**pwmoutput→load()****YPwmOutput**

Preloads the PWM cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→nextPwmOutput()pwmoutput→
nextPwmOutput ()

YPwmOutput

Continues the enumeration of PWMs started using `yFirstPwmOutput ()`.

`YPwmOutput * nextPwmOutput()`

Returns :

a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.

pwmoutput→**pulseDurationMove()****pwmoutput**→
pulseDurationMove()

YPwmOutput

Performs a smooth transistion of the pulse duration toward a given value.

```
int pulseDurationMove( double ms_target, int ms_duration)
```

Any period, frequency, duty cycle or pulse width change will cancel any ongoing transition process.

Parameters :

ms_target new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)

ms_duration total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→registerValueCallback()**pwmoutput→**
registerValueCallback()

YPwmOutput

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YPwmOutputValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmoutput→**set_dutyCycle()****YPwmOutput****pwmoutput**→**setDutyCycle()****pwmoutput**→**set_dutyCycle()**

Changes the PWM duty cycle, in per cents.

```
int set_dutyCycle( double newval)
```

Parameters :

newval a floating point number corresponding to the PWM duty cycle, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→**set_dutyCycleAtPowerOn()****YPwmOutput****pwmoutput**→**setDutyCycleAtPowerOn()****pwmoutput**→
set_dutyCycleAtPowerOn()

Changes the PWM duty cycle at device power on.

```
int set_dutyCycleAtPowerOn( double newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a floating point number corresponding to the PWM duty cycle at device power on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→**set_enabled()****YPwmOutput****pwmoutput**→**setEnabled()****pwmoutput**→**set_enabled()**

Stops or starts the PWM.

```
int set_enabled( Y_ENABLED_enum newval)
```

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabledAtPowerOn()**YPwmOutput****pwmoutput→setEnabledAtPowerOn()**
set_enabledAtPowerOn()

Changes the state of the PWM at device power on.

```
int set_enabledAtPowerOn( Y_ENABLEDATPOWERON_enum newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_frequency()**YPwmOutput****pwmoutput→setFrequency()**
pwmoutput→set_frequency()

Changes the PWM frequency.

```
int set_frequency( int newval)
```

The duty cycle is kept unchanged thanks to an automatic pulse width change.

Parameters :

newval an integer corresponding to the PWM frequency

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→**set_logicalName()****YPwmOutput****pwmoutput**→**setLogicalName()****pwmoutput**→
set_logicalName()

Changes the logical name of the PWM.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the PWM.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→set_period()

YPwmOutput

pwmoutput→setPeriod()`pwmoutput→set_period()`

Changes the PWM period.

```
int set_period( double newval)
```

Parameters :

newval a floating point number corresponding to the PWM period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_pulseDuration()**YPwmOutput****pwmoutput→setPulseDuration()**
pwmoutput→set_pulseDuration()

Changes the PWM pulse length, in milliseconds.

```
int set_pulseDuration( double newval)
```

A pulse length cannot be longer than period, otherwise it is truncated.

Parameters :

newval a floating point number corresponding to the PWM pulse length, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_userdata()

YPwmOutput

pwmoutput→setUserData()
pwmoutput→set_userdata()

Stores a user context provided as argument in the `userData` attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.31. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmpowersource.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPwmPowerSource = yoctolib.YPwmPowerSource;
php	require_once('yocto_pwmpowersource.php');
c++	#include "yocto_pwmpowersource.h"
m	#import "yocto_pwmpowersource.h"
pas	uses yocto_pwmpowersource;
vb	yocto_pwmpowersource.vb
cs	yocto_pwmpowersource.cs
java	import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py	from yocto_pwmpowersource import *

Global functions

yFindPwmPowerSource(func)

Retrieves a voltage source for a given identifier.

yFirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

YPwmPowerSource methods

pwmpowersource→describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form TYPE (NAME) = SERIAL . FUNCTIONID.

pwmpowersource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

pwmpowersource→get_errorMessage()

Returns the error message of the latest error with the voltage source.

pwmpowersource→get_errorType()

Returns the numerical error code of the latest error with the voltage source.

pwmpowersource→get_friendlyName()

Returns a global identifier of the voltage source in the format MODULE_NAME . FUNCTION_NAME.

pwmpowersource→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

pwmpowersource→get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

pwmpowersource→get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form SERIAL . FUNCTIONID.

pwmpowersource→get_logicalName()

Returns the logical name of the voltage source.

pwmpowersource→get_module()

Gets the YModule object for the device on which the function is located.

pwmpowersource→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

pwmpowersource→get_powerMode()

Returns the selected power source for the PWM on the same device

pwmpowersource→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

pwmpowersource→isOnline()

Checks if the voltage source is currently reachable, without raising any error.

pwmpowersource→isOnline_async(callback, context)

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

pwmpowersource→load(msValidity)

Preloads the voltage source cache with a specified validity duration.

pwmpowersource→load_async(msValidity, callback, context)

Preloads the voltage source cache with a specified validity duration (asynchronous version).

pwmpowersource→nextPwmPowerSource()

Continues the enumeration of Voltage sources started using yFirstPwmPowerSource().

pwmpowersource→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pwmpowersource→set_logicalName(newval)

Changes the logical name of the voltage source.

pwmpowersource→set_powerMode(newval)

Changes the PWM power source.

pwmpowersource→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmpowersource→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmPowerSource.FindPwmPowerSource() yFindPwmPowerSource()yFindPwmPowerSource()

YPwmPowerSource

Retrieves a voltage source for a given identifier.

```
YPwmPowerSource* yFindPwmPowerSource( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmPowerSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage source

Returns :

a `YPwmPowerSource` object allowing you to drive the voltage source.

**YPwmPowerSource.FirstPwmPowerSource()
yFirstPwmPowerSource()****yFirstPwmPowerSource()**

YPwmPowerSource

Starts the enumeration of Voltage sources currently accessible.

YPwmPowerSource* yFirstPwmPowerSource()

Use the method `YPwmPowerSource.nextPwmPowerSource()` to iterate on next Voltage sources.

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to the first source currently online, or a `null` pointer if there are none.

pwmpowersource→describe()pwmpowersource→
describe()

YPwmPowerSource

Returns a short text that describes unambiguously the instance of the voltage source in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage source (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmpowersource→get_advertisedValue()

YPwmPowerSource

pwmpowersource→advertisedValue()

pwmpowersource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the voltage source (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwmpowersource→get_errorMessage()**YPwmPowerSource****pwmpowersource→errorMessage()****pwmpowersource****→get_errorMessage()**

Returns the error message of the latest error with the voltage source.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the voltage source object

pwmpowersource→get_errorType()

YPwmPowerSource

pwmpowersource→errorType()
pwmpowersource→get_errorType()

Returns the numerical error code of the latest error with the voltage source.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the voltage source object

pwmpowersource→get_friendlyName()**YPwmPowerSource****pwmpowersource→friendlyName()****pwmpowersource****→get_friendlyName()**

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

```
string get_friendlyName( )
```

The returned string uses the logical names of the module and of the voltage source if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage source (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage source using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwmpowersource→get_functionDescriptor()

YPwmPowerSource

pwmpowersource→functionDescriptor()

pwmpowersource→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmpowersource→get_functionId()**YPwmPowerSource****pwmpowersource→functionId()**
pwmpowersource→get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

string **get_functionId()**

For example relay1

Returns :

a string that identifies the voltage source (ex: relay1) On failure, throws an exception or returns Y_FUNCTIONID_INVALID.

pwmpowersource→get_hardwareId()

YPwmPowerSource

pwmpowersource→hardwareId()
pwmpowersource→get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage source. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the voltage source (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

pwmpowersource→**get_logicalName()****YPwmPowerSource****pwmpowersource**→**logicalName()****pwmpowersource**
→**get_logicalName()**

Returns the logical name of the voltage source.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the voltage source. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pwmpowersource→get_module()

YPwmPowerSource

pwmpowersource→module()**pwmpowersource→**
get_module()

Gets the YModule object for the device on which the function is located.

```
YModule * get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

pwmpowersource→get_powerMode()**YPwmPowerSource****pwmpowersource→powerMode()**
pwmpowersource→get_powerMode()

Returns the selected power source for the PWM on the same device

[Y_POWERMODE_enum](#) **get_powerMode()**

Returns :

a value among Y_POWERMODE_USB_5V, Y_POWERMODE_USB_3V, Y_POWERMODE_EXT_V and Y_POWERMODE_OPNDRN corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns Y_POWERMODE_INVALID.

pwmpowersource→get_userData()

YPwmPowerSource

pwmpowersource→userData()
pwmpowersource→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
void * get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

pwmpowersource→isOnline()**pwmpowersource→isOnline()**

YPwmPowerSource

Checks if the voltage source is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

Returns :

`true` if the voltage source can be reached, and `false` otherwise

pwmpowersource→**load()****pwmpowersource**→**load()****YPwmPowerSource**

Preloads the voltage source cache with a specified validity duration.

YRETCODE **load(int msValidity)**

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmpowersource→nextPwmPowerSource()**YPwmPowerSource****pwmpowersource→nextPwmPowerSource()**

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

`YPwmPowerSource * nextPwmPowerSource()`

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more Voltage sources to enumerate.

pwmpowersource→registerValueCallback()

YPwmPowerSource

pwmpowersource→registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YPwmPowerSourceValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmpowersource→set_logicalName()**YPwmPowerSource****pwmpowersource→setLogicalName()****pwmpowersource→set_logicalName()**

Changes the logical name of the voltage source.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage source.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

pwmpowersource→set_powerMode()**YPwmPowerSource****pwmpowersource→setPowerMode()****pwmpowersource→set_powerMode()**

Changes the PWM power source.

```
int set_powerMode( Y_POWERMODE_enum newval)
```

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

Parameters :

newval a value among Y_POWERMODE_USB_5V, Y_POWERMODE_USB_3V, Y_POWERMODE_EXT_V and Y_POWERMODE_OPNDRN corresponding to the PWM power source

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→set_userdata()**YPwmPowerSource****pwmpowersource→setUserData()****pwmpowersource→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.32. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YGyro = yoctolib.YGyro;
php	require_once('yocto_gyro.php');
c++	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

Global functions

yFindQt(func)

Retrieves a quaternion component for a given identifier.

yFirstQt()

Starts the enumeration of quaternion components currently accessible.

YQt methods

qt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

qt→describe()

Returns a short text that describes unambiguously the instance of the quaternion component in the form TYPE (NAME) =SERIAL . FUNCTIONID.

qt→get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

qt→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

qt→get_currentValue()

Returns the current value of the value.

qt→get_errorMessage()

Returns the error message of the latest error with the quaternion component.

qt→get_errorType()

Returns the numerical error code of the latest error with the quaternion component.

qt→get_friendlyName()

Returns a global identifier of the quaternion component in the format MODULE_NAME . FUNCTION_NAME.

qt→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

qt→get_functionId()

Returns the hardware identifier of the quaternion component, without reference to the module.

qt→get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

qt→get_highestValue()

Returns the maximal value observed for the value since the device was started.

qt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

qt→get_logicalName()

Returns the logical name of the quaternion component.

qt→get_lowestValue()

Returns the minimal value observed for the value since the device was started.

qt→get_module()

Gets the `YModule` object for the device on which the function is located.

qt→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

qt→get_recordedData(startTime, endTime)

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

qt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

qt→get_resolution()

Returns the resolution of the measured values.

qt→get_unit()

Returns the measuring unit for the value.

qt→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

qt→isOnline()

Checks if the quaternion component is currently reachable, without raising any error.

qt→isOnline_async(callback, context)

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

qt→load(msValidity)

Preloads the quaternion component cache with a specified validity duration.

qt→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

qt→load_async(msValidity, callback, context)

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

qt→nextQt()

Continues the enumeration of quaternion components started using `yFirstQt()`.

qt→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

qt→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

qt→set_highestValue(newval)

Changes the recorded maximal value observed.

qt→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

qt→set_logicalName(newval)

3. Reference

Changes the logical name of the quaternion component.

qt→set_lowestValue(newval)

Changes the recorded minimal value observed.

qt→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

qt→set_resolution(newval)

Changes the resolution of the measured physical values.

qt→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

qt→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YQt.FindQt()**YQt****yFindQt()****yFindQt ()**

Retrieves a quaternion component for a given identifier.

```
YQt* yFindQt( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQt.isOnline()` to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the quaternion component

Returns :

a `YQt` object allowing you to drive the quaternion component.

YQt.FirstQt()

YQt

yFirstQt()`yFirstQt()`

Starts the enumeration of quaternion components currently accessible.

`YQt* yFirstQt()`

Use the method `YQt.nextQt()` to iterate on next quaternion components.

Returns :

a pointer to a `YQt` object, corresponding to the first quaternion component currently online, or a `null` pointer if there are none.

qt→calibrateFromPoints()qt→
calibrateFromPoints()

YQt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→describe()**qt→describe()****YQt**

Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

`string describe()`

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the quaternion component (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

qt→get_advertisedValue()**YQt****qt→advertisedValue()****qt→get_advertisedValue()**

Returns the current value of the quaternion component (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the quaternion component (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

qt→get_currentRawValue()

YQt

qt→currentRawValue()qt→

get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

`double get_currentRawValue()`

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

qt→get_currentValue()**YQt****qt→currentValue()****qt→get_currentValue()**

Returns the current value of the value.

```
double get_currentValue( )
```

Returns :

a floating point number corresponding to the current value of the value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

qt→get_errorMessage()

YQt

qt→errorMessage()qt→get_errorMessage()

Returns the error message of the latest error with the quaternion component.

string **get_errorMessage**()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the quaternion component object

qt→get_errorType()**YQt****qt→errorType()****qt→get_errorType()**

Returns the numerical error code of the latest error with the quaternion component.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the quaternion component object

qt→get_friendlyName()**YQt****qt→friendlyName()****qt→get_friendlyName()**

Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the quaternion component if they are defined, otherwise the serial number of the module and the hardware identifier of the quaternion component (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the quaternion component using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

qt→get_functionDescriptor()
qt→functionDescriptor()qt→
get_functionDescriptor()

YQt

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

qt→get_functionId()

YQt

qt→functionId()`qt→get_functionId()`

Returns the hardware identifier of the quaternion component, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the quaternion component (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

qt→get_hardwareId()**YQt****qt→hardwareId()****qt→get_hardwareId()**

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quaternion component. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the quaternion component (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

qt→get_highestValue()

YQt

qt→highestValue() **qt→get_highestValue()**

Returns the maximal value observed for the value since the device was started.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

qt→get_logFrequency()**YQt****qt→logFrequency()****qt→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

qt→get_logicalName()

YQt

qt→logicalName()qt→get_logicalName()

Returns the logical name of the quaternion component.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the quaternion component. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

qt→get_lowestValue()**YQt****qt→lowestValue()**qt→get_lowestValue()

Returns the minimal value observed for the value since the device was started.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the value since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

qt→get_module()

YQt

qt→module()qt→get_module()

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

qt→get_recordedData()**YQt****qt→recordedData()****qt→get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

qt→get_reportFrequency()

YQt

qt→reportFrequency()qt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

qt→get_resolution()**YQt****qt→resolution()****qt→get_resolution()**

Returns the resolution of the measured values.

`double get_resolution()`

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

qt→get_unit()

YQt

qt→unit()qt→get_unit()

Returns the measuring unit for the value.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the value

On failure, throws an exception or returns Y_UNIT_INVALID.

qt→get_userdata()**YQt****qt→userdata()**qt→get_userdata()

Returns the value of the userData attribute, as previously stored using method set_userdata.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

qt→isOnline()**qt→isOnline()****YQt**

Checks if the quaternion component is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

Returns :

`true` if the quaternion component can be reached, and `false` otherwise

qt→load()**qt→load()****YQt**

Preloads the quaternion component cache with a specified validity duration.

YRETCODE load(int msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

qt→loadCalibrationPoints()qt→
loadCalibrationPoints()

YQt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→**nextQt()****qt**→**nextQt()****YQt**

Continues the enumeration of quaternion components started using `yFirstQt()`.

`YQt * nextQt()`

Returns :

a pointer to a `YQt` object, corresponding to a quaternion component currently online, or a `null` pointer if there are no more quaternion components to enumerate.

qt→registerTimedReportCallback()qt→
registerTimedReportCallback()

YQt

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YQtTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

qt→registerValueCallback()qt→
registerValueCallback()

YQt

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YQtValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

qt→set_highestValue()**YQt****qt→setHighestValue()****qt→set_highestValue()**

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logFrequency()**YQt****qt→setLogFrequency()****qt→set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logicalName()**YQt****qt→setLogicalName()****qt→set_logicalName()**

Changes the logical name of the quaternion component.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the quaternion component.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

qt→set_lowestValue()**YQt****qt→setLowestValue()****qt→set_lowestValue()**

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_reportFrequency()**YQt****qt→setReportFrequency()**qt→**set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_resolution()**YQt****qt→setResolution()****qt→set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_userdata()

YQt

qt→setUserData() **qt→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.33. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_realtimeclock.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YRealTimeClock = yoctolib.YRealTimeClock;
php	require_once('yocto_realtimeclock.php');
c++	#include "yocto_realtimeclock.h"
m	#import "yocto_realtimeclock.h"
pas	uses yocto_realtimeclock;
vb	yocto_realtimeclock.vb
cs	yocto_realtimeclock.cs
java	import com.yoctopuce.YoctoAPI.YRealTimeClock;
py	from yocto_realtimeclock import *

Global functions

yFindRealTimeClock(func)

Retrieves a clock for a given identifier.

yFirstRealTimeClock()

Starts the enumeration of clocks currently accessible.

YRealTimeClock methods

realtimeclock→describe()

Returns a short text that describes unambiguously the instance of the clock in the form TYPE (NAME) = SERIAL . FUNCTIONID.

realtimeclock→get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

realtimeclock→get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

realtimeclock→get_errorMessage()

Returns the error message of the latest error with the clock.

realtimeclock→get_errorType()

Returns the numerical error code of the latest error with the clock.

realtimeclock→get_friendlyName()

Returns a global identifier of the clock in the format MODULE_NAME . FUNCTION_NAME.

realtimeclock→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

realtimeclock→get_functionId()

Returns the hardware identifier of the clock, without reference to the module.

realtimeclock→get_hardwareId()

Returns the unique hardware identifier of the clock in the form SERIAL . FUNCTIONID.

realtimeclock→get_logicalName()

Returns the logical name of the clock.

realtimeclock→get_module()

	Gets the <code>YModule</code> object for the device on which the function is located.
<code>realtimeclock→get_module_async(callback, context)</code>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<code>realtimeclock→get_timeSet()</code>	Returns true if the clock has been set, and false otherwise.
<code>realtimeclock→get_unixTime()</code>	Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).
<code>realtimeclock→get_userData()</code>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<code>realtimeclock→get_utcOffset()</code>	Returns the number of seconds between current time and UTC time (time zone).
<code>realtimeclock→isOnline()</code>	Checks if the clock is currently reachable, without raising any error.
<code>realtimeclock→isOnline_async(callback, context)</code>	Checks if the clock is currently reachable, without raising any error (asynchronous version).
<code>realtimeclock→load(msValidity)</code>	Preloads the clock cache with a specified validity duration.
<code>realtimeclock→load_async(msValidity, callback, context)</code>	Preloads the clock cache with a specified validity duration (asynchronous version).
<code>realtimeclock→nextRealTimeClock()</code>	Continues the enumeration of clocks started using <code>yFirstRealTimeClock()</code> .
<code>realtimeclock→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>realtimeclock→set_logicalName(newval)</code>	Changes the logical name of the clock.
<code>realtimeclock→set_unixTime(newval)</code>	Changes the current time.
<code>realtimeclock→set_userData(data)</code>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<code>realtimeclock→set_utcOffset(newval)</code>	Changes the number of seconds between current time and UTC time (time zone).
<code>realtimeclock→wait_async(callback, context)</code>	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRealTimeClock.FindRealTimeClock() **yFindRealTimeClock()****yFindRealTimeClock()**

YRealTimeClock

Retrieves a clock for a given identifier.

```
YRealTimeClock* yFindRealTimeClock( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the clock is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRealTimeClock.isOnline()` to test if the clock is indeed online at a given time. In case of ambiguity when looking for a clock by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the clock

Returns :

a `YRealTimeClock` object allowing you to drive the clock.

YRealTimeClock.FirstRealTimeClock()

YRealTimeClock

yFirstRealTimeClock()`yFirstRealTimeClock()`

Starts the enumeration of clocks currently accessible.

`YRealTimeClock* yFirstRealTimeClock()`

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

Returns :

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a `null` pointer if there are none.

realtimeclock→describe()**realtimeclock→
describe()**

YRealTimeClock

Returns a short text that describes unambiguously the instance of the clock in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the clock (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

realtimeclock→get_advertisedValue()

YRealTimeClock

realtimeclock→advertisedValue()**realtimeclock→**
get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the clock (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

realtimeclock→get_dateTime()**YRealTimeClock****realtimeclock→dateTime()****realtimeclock→**
get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

string **get_dateTime()**

Returns :

a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns Y_DATETIME_INVALID.

realtimeclock→get_errorMessage()

YRealTimeClock

realtimeclock→errorMessage()**realtimeclock→**

get_errorMessage()

Returns the error message of the latest error with the clock.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the clock object

realtimeclock→get_errorType()**YRealTimeClock****realtimeclock→errorType()****realtimeclock→**
get_errorType()

Returns the numerical error code of the latest error with the clock.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the clock object

realtimeclock→get_friendlyName()

YRealTimeClock

realtimeclock→friendlyName()**realtimeclock→**
get_friendlyName()

Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the clock if they are defined, otherwise the serial number of the module and the hardware identifier of the clock (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the clock using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

realtimeclock→get_functionDescriptor()**YRealTimeClock****realtimeclock→functionDescriptor()realtimeclock****→get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

realtimeclock→get_functionId()

YRealTimeClock

realtimeclock→functionId()**realtimeclock→**
get_functionId()

Returns the hardware identifier of the clock, without reference to the module.

string get_functionId()

For example relay1

Returns :

a string that identifies the clock (ex: relay1) On failure, throws an exception or returns Y_FUNCTIONID_INVALID.

realtimeclock→get_hardwareId()**YRealTimeClock****realtimeclock→hardwareId()****realtimeclock→****get_hardwareId()**

Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.

```
string get_hardwareId()
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the clock. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the clock (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

realtimeclock→get_logicalName()

YRealTimeClock

realtimeclock→logicalName()**realtimeclock→**
get_logicalName()

Returns the logical name of the clock.

string get_logicalName()

Returns :

a string corresponding to the logical name of the clock. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

realtimeclock→get_module()**YRealTimeClock****realtimeclock→module()****realtimeclock→**
get_module()

Gets the YModule object for the device on which the function is located.

```
YModule * get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

realtimeclock→get_timeSet()

YRealTimeClock

realtimeclock→timeSet()**realtimeclock→**
get_timeSet()

Returns true if the clock has been set, and false otherwise.

Y_TIMESET_enum **get_timeSet()**

Returns :

either Y_TIMESET_FALSE or Y_TIMESET_TRUE, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns Y_TIMESET_INVALID.

realtimeclock→**get_unixTime()****YRealTimeClock****realtimeclock**→**unixTime()****realtimeclock**→**get_unixTime()**

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

s64 **get_unixTime()**

Returns :

an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns Y_UNIXTIME_INVALID.

realtimeclock→get_userdata()

YRealTimeClock

realtimeclock→userdata()**realtimeclock→**
get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

realtimeclock→**get_utcOffset()****YRealTimeClock****realtimeclock**→**utcOffset()****realtimeclock**→
get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

int **get_utcOffset()**

Returns :

an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns Y_UTC_OFFSET_INVALID.

realtimeclock→isOnline()**realtimeclock→**
isOnline()

YRealTimeClock

Checks if the clock is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

Returns :

`true` if the clock can be reached, and `false` otherwise

realtimeclock→load()**realtimeclock→load()****YRealTimeClock**

Preloads the clock cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

realtimeclock→**nextRealTimeClock()****realtimeclock** **YRealTimeClock**
→**nextRealTimeClock()**

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

`YRealTimeClock * nextRealTimeClock()`

Returns :

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a null pointer if there are no more clocks to enumerate.

realtimeclock→registerValueCallback()**YRealTimeClock****realtimeclock→registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YRealTimeClockValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

realtimeclock→set_logicalName()**YRealTimeClock****realtimeclock→setLogicalName()****realtimeclock→set_logicalName()**

Changes the logical name of the clock.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the clock.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

realtimeclock→set_unixTime()**YRealTimeClock****realtimeclock→setUnixTime()**
realtimeclock→set_unixTime()

Changes the current time.

```
int set_unixTime( s64 newval)
```

Time is specified in Unix format (number of elapsed seconds since Jan 1st, 1970). If current UTC time is known, utcOffset will be automatically adjusted for the new specified time.

Parameters :

newval an integer corresponding to the current time

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→set_userdata()

YRealTimeClock

realtimeclock→setUserData()**realtimeclock→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

realtimeclock→set_utcOffset()**YRealTimeClock****realtimeclock→setUtcOffset()**
realtimeclock→set_utcOffset()

Changes the number of seconds between current time and UTC time (time zone).

```
int set_utcOffset( int newval)
```

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time will automatically be updated according to the selected time zone.

Parameters :

newval an integer corresponding to the number of seconds between current time and UTC time (time zone)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.34. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_refframe.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YRefFrame = yoctolib.YRefFrame;
php	require_once('yocto_refframe.php');
c++	#include "yocto_refframe.h"
m	#import "yocto_refframe.h"
pas	uses yocto_refframe;
vb	yocto_refframe.vb
cs	yocto_refframe.cs
java	import com.yoctopuce.YoctoAPI.YRefFrame;
py	from yocto_refframe import *

Global functions

yFindRefFrame(func)

Retrieves a reference frame for a given identifier.

yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

YRefFrame methods

refframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

refframe→describe()

Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE (NAME) = SERIAL . FUNCTIONID.

refframe→get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method start3DCalibration.

refframe→get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

refframe→get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method start3DCalibration.

refframe→get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method start3DCalibration.

refframe→get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method start3DCalibration.

refframe→get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

refframe→get_bearing()

Returns the reference bearing used by the compass.

refframe→get_errorMessage()

Returns the error message of the latest error with the reference frame.

refframe→get_errorType()

Returns the numerical error code of the latest error with the reference frame.

refframe→get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME . FUNCTION_NAME`.

refframe→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

refframe→get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

refframe→get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form `SERIAL . FUNCTIONID`.

refframe→get_logicalName()

Returns the logical name of the reference frame.

refframe→get_module()

Gets the `YModule` object for the device on which the function is located.

refframe→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

refframe→get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

refframe→get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

refframe→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

refframe→isOnline()

Checks if the reference frame is currently reachable, without raising any error.

refframe→isOnline_async(callback, context)

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

refframe→load(msValidity)

Preloads the reference frame cache with a specified validity duration.

refframe→load_async(msValidity, callback, context)

Preloads the reference frame cache with a specified validity duration (asynchronous version).

refframe→more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

refframe→nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

refframe→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

refframe→save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

refframe→set_bearing(newval)

Changes the reference bearing used by the compass.

refframe→set_logicalName(newval)

3. Reference

Changes the logical name of the reference frame.

reframe→**set_mountPosition**(**position**, **orientation**)

Changes the compass and tilt sensor frame of reference.

reframe→**set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

reframe→**start3DCalibration**()

Initiates the sensors tridimensional calibration process.

reframe→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRefFrame.FindRefFrame()**YRefFrame****yFindRefFrame()****yFindRefFrame()**

Retrieves a reference frame for a given identifier.

```
YRefFrame* yFindRefFrame( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the reference frame

Returns :

a `YRefFrame` object allowing you to drive the reference frame.

YRefFrame.FirstRefFrame()

YRefFrame

yFirstRefFrame()**yFirstRefFrame()**

Starts the enumeration of reference frames currently accessible.

`YRefFrame* yFirstRefFrame()`

Use the method `YRefFrame.nextRefFrame()` to iterate on next reference frames.

Returns :

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

refframe→cancel3DCalibration()refframe→
cancel3DCalibration()

YRefFrame

Aborts the sensors tridimensional calibration process et restores normal settings.

int cancel3DCalibration()

On failure, throws an exception or returns a negative error code.

refframe→describe()**refframe→describe()****YRefFrame**

Returns a short text that describes unambiguously the instance of the reference frame in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the reference frame (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

refframe→get_3DCalibrationHint()**YRefFrame****refframe→3DCalibrationHint()****refframe→****get_3DCalibrationHint()**

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

string **get_3DCalibrationHint()**

Returns :

a character string.

refframe→get_3DCalibrationLogMsg()

YRefFrame

refframe→3DCalibrationLogMsg()refframe→

get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

string [get_3DCalibrationLogMsg\(\)](#)

When no new message is available, returns an empty string.

Returns :

a character string.

refframe→get_3DCalibrationProgress()**YRefFrame****refframe→3DCalibrationProgress()****refframe→**
get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

int **get_3DCalibrationProgress()**

Returns :

an integer between 0 (not started) and 100 (stage completed).

refframe→**get_3DCalibrationStage()**

YRefFrame

refframe→**3DCalibrationStage()****refframe**→

get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

int **get_3DCalibrationStage()**

Returns :

an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress()	YRefFrame
refframe→3DCalibrationStageProgress()	refframe→
get_3DCalibrationStageProgress()	

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

int get_3DCalibrationStageProgress()

Returns :

an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue()

YRefFrame

refframe→advertisedValue()**refframe→**

get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the reference frame (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

refframe→**get_bearing()****YRefFrame****refframe**→**bearing()****refframe**→**get_bearing()**

Returns the reference bearing used by the compass.

double **get_bearing()**

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

Returns :

a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns Y_BEARING_INVALID.

refframe→get_errorMessage()

YRefFrame

refframe→errorMessage()**refframe→**

get_errorMessage()

Returns the error message of the latest error with the reference frame.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the reference frame object

refframe→get_errorType()**YRefFrame****refframe→errorType()****refframe→get_errorType()**

Returns the numerical error code of the latest error with the reference frame.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the reference frame object

refframe→**get_friendlyName()**

YRefFrame

refframe→**friendlyName()****refframe**→

get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the reference frame if they are defined, otherwise the serial number of the module and the hardware identifier of the reference frame (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the reference frame using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

refframe→**get_functionDescriptor()****YRefFrame****refframe**→**functionDescriptor()****refframe**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

refframe→**get_functionId()**

YRefFrame

refframe→**functionId()****refframe**→

get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

string **get_functionId()** ()

For example `relay1`

Returns :

a string that identifies the reference frame (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

refframe→get_hardwareId()**YRefFrame****refframe→hardwareId()****refframe→****get_hardwareId()**

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the reference frame (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

refframe→**get_logicalName()**

YRefFrame

refframe→**logicalName()****refframe**→

get_logicalName()

Returns the logical name of the reference frame.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the reference frame. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

refframe→get_module()**YRefFrame****refframe→module()****refframe→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

refframe→**get_mountOrientation()****YRefFrame****refframe**→**mountOrientation()****refframe**→**get_mountOrientation()**

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

Y_MOUNTORIENTATION **get_mountOrientation()****Returns :**

a value among the enumeration **Y_MOUNTORIENTATION** (**Y_MOUNTORIENTATION_TWELVE**, **Y_MOUNTORIENTATION_THREE**, **Y_MOUNTORIENTATION_SIX**, **Y_MOUNTORIENTATION_NINE**) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

refframe→get_mountPosition()**YRefFrame****refframe→mountPosition()****refframe→**
get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

Y_MOUNTPOSITION **get_mountPosition()**

Returns :

a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.

refframe→get_userdata()

YRefFrame

refframe→userdata()**refframe→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

refframe→isOnline()**refframe→isOnline()****YRefFrame**

Checks if the reference frame is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

Returns :

`true` if the reference frame can be reached, and `false` otherwise

refframe→load()**refframe→load()****YRefFrame**

Preloads the reference frame cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

refframe→**more3DCalibration()****refframe**→
more3DCalibration()

YRefFrame

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

int **more3DCalibration()**

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.

refframe→**nextRefFrame()****refframe**→
nextRefFrame()

YRefFrame

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

`YRefFrame * nextRefFrame()`

Returns :

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a `null` pointer if there are no more reference frames to enumerate.

refframe→**registerValueCallback()****refframe**→
registerValueCallback()

YRefFrame

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YRefFrameValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

refframe→**save3DCalibration()****refframe**→
save3DCalibration()

YRefFrame

Applies the sensors tridimensional calibration parameters that have just been computed.

int **save3DCalibration()**

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

refframe→set_bearing()**YRefFrame****refframe→setBearing()****refframe→set_bearing()**

Changes the reference bearing used by the compass.

```
int set_bearing( double newval)
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→**set_logicalName()****YRefFrame****refframe**→**setLogicalName()****refframe**→
set_logicalName()

Changes the logical name of the reference frame.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the reference frame.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

refframe→set_mountPosition()**YRefFrame****refframe→setMountPosition()refframe→****set_mountPosition()**

Changes the compass and tilt sensor frame of reference.

```
int set_mountPosition( Y_MOUNTPOSITION position,
                      Y_MOUNTORIENTATION orientation)
```

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

Parameters :

position a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

orientation a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

refframe→set_userdata()

YRefFrame

refframe→setUserData()**refframe→**

set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

refframe→start3DCalibration()**refframe→**
start3DCalibration()

YRefFrame

Initiates the sensors tridimensional calibration process.

int start3DCalibration()

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

3.35. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_relay.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YRelay = yoctolib.YRelay;</code>
php	<code>require_once('yocto_relay.php');</code>
c++	<code>#include "yocto_relay.h"</code>
m	<code>#import "yocto_relay.h"</code>
pas	<code>uses yocto_relay;</code>
vb	<code>yocto_relay.vb</code>
cs	<code>yocto_relay.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YRelay;</code>
py	<code>from yocto_relay import *</code>

Global functions

yFindRelay(func)

Retrieves a relay for a given identifier.

yFirstRelay()

Starts the enumeration of relays currently accessible.

YRelay methods

relay→delayedPulse(ms_delay, ms_duration)

Schedules a pulse.

relay→describe()

Returns a short text that describes unambiguously the instance of the relay in the form
TYPE (NAME) =SERIAL . FUNCTIONID.

relay→get_advertisedValue()

Returns the current value of the relay (no more than 6 characters).

relay→get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

relay→get_errorMessage()

Returns the error message of the latest error with the relay.

relay→get_errorType()

Returns the numerical error code of the latest error with the relay.

relay→get_friendlyName()

Returns a global identifier of the relay in the format MODULE_NAME . FUNCTION_NAME.

relay→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

relay→get_functionId()

Returns the hardware identifier of the relay, without reference to the module.

relay→get_hardwareId()

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

relay→get_logicalName()

Returns the logical name of the relay.

relay→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state A before automatically switching back in to B state.

relay→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state B before automatically switching back in to A state.

relay→get_module()

Gets the `YModule` object for the device on which the function is located.

relay→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

relay→get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

relay→get_pulseTimer()

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

relay→get_state()

Returns the state of the relays (A for the idle position, B for the active position).

relay→get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

relay→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

relay→isOnline()

Checks if the relay is currently reachable, without raising any error.

relay→isOnline_async(callback, context)

Checks if the relay is currently reachable, without raising any error (asynchronous version).

relay→load(msValidity)

Preloads the relay cache with a specified validity duration.

relay→load_async(msValidity, callback, context)

Preloads the relay cache with a specified validity duration (asynchronous version).

relay→nextRelay()

Continues the enumeration of relays started using `yFirstRelay()`.

relay→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

relay→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

relay→set_logicalName(newval)

Changes the logical name of the relay.

relay→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state A before automatically switching back in to B state.

relay→set_maxTimeOnStateB(newval)

3. Reference

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

relay→set_output(newval)

Changes the output state of the relays, when used as a simple switch (single throw).

relay→set_state(newval)

Changes the state of the relays (A for the idle position, B for the active position).

relay→set_stateAtPowerOn(newval)

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

relay→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

relay→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRelay.FindRelay() yFindRelay()yFindRelay()

YRelay

Retrieves a relay for a given identifier.

```
YRelay* yFindRelay( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the relay is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRelay.isOnline()` to test if the relay is indeed online at a given time. In case of ambiguity when looking for a relay by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the relay

Returns :

a `YRelay` object allowing you to drive the relay.

YRelay.FirstRelay()

YRelay

yFirstRelay()**yFirstRelay()**

Starts the enumeration of relays currently accessible.

YRelay* **yFirstRelay()**

Use the method `YRelay.nextRelay()` to iterate on next relays.

Returns :

a pointer to a `YRelay` object, corresponding to the first relay currently online, or a `null` pointer if there are none.

relay→delayedPulse()**relay→delayedPulse()**

YRelay

Schedules a pulse.

```
int delayedPulse( int ms_delay, int ms_duration)
```

Parameters :

ms_delay waiting time before the pulse, in milliseconds

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→describe()**relay→describe()****YRelay**

Returns a short text that describes unambiguously the instance of the relay in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the relay (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

relay→**get_advertisedValue()**
relay→**advertisedValue()****relay**→
get_advertisedValue()

YRelay

Returns the current value of the relay (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the relay (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

relay→get_countdown()

YRelay

relay→countdown()**relay→get_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

s64 **get_countdown()**

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns Y_COUNTDOWN_INVALID.

relay→get_errorMessage()**YRelay****relay→errorMessage()****relay→get_errorMessage()**

Returns the error message of the latest error with the relay.

```
string get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the relay object

relay→get_errorType()

YRelay

relay→errorType()**relay→get_errorType()**

Returns the numerical error code of the latest error with the relay.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the relay object

relay→get_friendlyName()**YRelay****relay→friendlyName()****relay→get_friendlyName()**

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the relay if they are defined, otherwise the serial number of the module and the hardware identifier of the relay (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the relay using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

relay→**get_functionDescriptor()**

YRelay

relay→**functionDescriptor()****relay**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

relay→get_functionId()**YRelay****relay→functionId()****relay→get_functionId()**

Returns the hardware identifier of the relay, without reference to the module.

```
string get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the relay (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

relay→get_hardwareId()

YRelay

relay→hardwareId()**relay→get_hardwareId()**

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the relay. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the relay (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

relay→get_logicalName()**YRelay****relay→logicalName()****relay→get_logicalName()**

Returns the logical name of the relay.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the relay. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

relay→get_maxTimeOnStateA()

YRelay

relay→maxTimeOnStateA()
relay→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

s64 **get_maxTimeOnStateA()**

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.

relay→get_maxTimeOnStateB()**YRelay****relay→maxTimeOnStateB()**
get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

s64 **get_maxTimeOnStateB()**

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEB_INVALID.

relay→get_module()

YRelay

relay→module()**relay→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

relay→get_output()**YRelay****relay→output()**~~**relay→get_output()**~~

Returns the output state of the relays, when used as a simple switch (single throw).

`Y_OUTPUT_enum` **get_output()** ()

Returns :

either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns `Y_OUTPUT_INVALID`.

relay→get_pulseTimer()

YRelay

relay→pulseTimer()**relay→get_pulseTimer()**

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

s64 **get_pulseTimer()**

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

relay→get_state()**YRelay****relay→state()****relay→get_state()**

Returns the state of the relays (A for the idle position, B for the active position).

Y_STATE_enum **get_state()**

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

relay→get_stateAtPowerOn()

YRelay

relay→stateAtPowerOn()**relay→**
get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

[Y_STATEATPOWERON_enum](#) **get_stateAtPowerOn()**

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

relay→get_userdata()**YRelay****relay→userdata()relay→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

relay→isOnline()**relay→isOnline()**

YRelay

Checks if the relay is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the relay.

Returns :

`true` if the relay can be reached, and `false` otherwise

relay→load()**relay→load()****YRelay**

Preloads the relay cache with a specified validity duration.

YRETCODE load(int msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

relay→nextRelay()`relay→nextRelay()`

YRelay

Continues the enumeration of relays started using `yFirstRelay()`.

`YRelay * nextRelay()`

Returns :

a pointer to a `YRelay` object, corresponding to a relay currently online, or a `null` pointer if there are no more relays to enumerate.

relay→pulse()**relay→pulse()****YRelay**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
int pulse( int ms_duration)
```

Parameters :

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**registerValueCallback()****relay**→
registerValueCallback()

YRelay

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YRelayValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

relay→**set_logicalName()****YRelay****relay**→**setLogicalName()****relay**→
set_logicalName()

Changes the logical name of the relay.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the relay.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

relay→**set_maxTimeOnStateA()****YRelay****relay**→**setMaxTimeOnStateA()****relay**→
set_maxTimeOnStateA()

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
int set_maxTimeOnStateA( s64 newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**set_maxTimeOnStateB()****YRelay****relay**→**setMaxTimeOnStateB()****relay**→**set_maxTimeOnStateB()**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
int set_maxTimeOnStateB( s64 newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_output()

YRelay

relay→setOutput()**relay→set_output()**

Changes the output state of the relays, when used as a simple switch (single throw).

```
int set_output( Y_OUTPUT_enum newval)
```

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_state()**YRelay****relay→setState()****relay→set_state()**

Changes the state of the relays (A for the idle position, B for the active position).

```
int set_state( Y_STATE_enum newval)
```

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**set_stateAtPowerOn()****YRelay****relay**→**setStateAtPowerOn()****relay**→
set_stateAtPowerOn()

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
int set_stateAtPowerOn( Y_STATEATPOWERON_enum newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_userdata()**YRelay****relay→setUserData()****relay→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.36. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

Global functions

yFindSensor(**func**)

Retrieves a sensor for a given identifier.

yFirstSensor()

Starts the enumeration of sensors currently accessible.

YSensor methods

sensor→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

sensor→describe()

Returns a short text that describes unambiguously the instance of the sensor in the form
TYPE (NAME) = SERIAL . FUNCTIONID.

sensor→get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

sensor→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

sensor→get_currentValue()

Returns the current value of the measure.

sensor→get_errorMessage()

Returns the error message of the latest error with the sensor.

sensor→get_errorType()

Returns the numerical error code of the latest error with the sensor.

sensor→get_friendlyName()

Returns a global identifier of the sensor in the format MODULE_NAME . FUNCTION_NAME.

sensor→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

sensor→get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

sensor→get_hardwareId()

Returns the unique hardware identifier of the sensor in the form `SERIAL . FUNCTIONID`.

sensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

sensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

sensor→get_logicalName()

Returns the logical name of the sensor.

sensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

sensor→get_module()

Gets the `YModule` object for the device on which the function is located.

sensor→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

sensor→get_recordedData(startTime, endTime)

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

sensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

sensor→get_resolution()

Returns the resolution of the measured values.

sensor→get_unit()

Returns the measuring unit for the measure.

sensor→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

sensor→isOnline()

Checks if the sensor is currently reachable, without raising any error.

sensor→isOnline_async(callback, context)

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

sensor→load(msValidity)

Preloads the sensor cache with a specified validity duration.

sensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

sensor→load_async(msValidity, callback, context)

Preloads the sensor cache with a specified validity duration (asynchronous version).

sensor→nextSensor()

Continues the enumeration of sensors started using `yFirstSensor()`.

sensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

sensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

sensor→set_highestValue(newval)

Changes the recorded maximal value observed.

sensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

sensor→set_logicalName(newval)

3. Reference

Changes the logical name of the sensor.

sensor→**set_lowestValue**(**newval**)

Changes the recorded minimal value observed.

sensor→**set_reportFrequency**(**newval**)

Changes the timed value notification frequency for this function.

sensor→**set_resolution**(**newval**)

Changes the resolution of the measured physical values.

sensor→**set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

sensor→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YSensor.FindSensor()**YSensor****yFindSensor()****yFindSensor()**

Retrieves a sensor for a given identifier.

```
YSensor* yFindSensor( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the sensor

Returns :

a `YSensor` object allowing you to drive the sensor.

YSensor.FirstSensor()

YSensor

yFirstSensor()**yFirstSensor()**

Starts the enumeration of sensors currently accessible.

YSensor* **yFirstSensor()**

Use the method `YSensor.nextSensor()` to iterate on next sensors.

Returns :

a pointer to a `YSensor` object, corresponding to the first sensor currently online, or a `null` pointer if there are none.

sensor→calibrateFromPoints() **sensor→**
calibrateFromPoints()

YSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→describe()**sensor→describe()****YSensor**

Returns a short text that describes unambiguously the instance of the sensor in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

sensor→**get_advertisedValue()****YSensor****sensor**→**advertisedValue()****sensor**→
get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

sensor→**get_currentRawValue()**

YSensor

sensor→**currentRawValue()****sensor**→

get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

`double get_currentRawValue()`

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

sensor→**get_currentValue()****YSensor****sensor**→**currentValue()****sensor**→
get_currentValue()

Returns the current value of the measure.

`double` **get_currentValue()**

Returns :

a floating point number corresponding to the current value of the measure

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

sensor→**get_errorMessage()**

YSensor

sensor→**errorMessage()****sensor**→

get_errorMessage()

Returns the error message of the latest error with the sensor.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the sensor object

sensor→get_errorType()**YSensor****sensor→errorType()****sensor→get_errorType()**

Returns the numerical error code of the latest error with the sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the sensor object

sensor→**get_friendlyName()**

YSensor

sensor→**friendlyName()****sensor**→
get_friendlyName()

Returns a global identifier of the sensor in the format `MODULE_NAME . FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

sensor→**get_functionDescriptor()**
sensor→**functionDescriptor()****sensor**→
get_functionDescriptor()

YSensor

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

sensor→**get_functionId()**

YSensor

sensor→**functionId()****sensor**→**get_functionId()**

Returns the hardware identifier of the sensor, without reference to the module.

string **get_functionId()** ()

For example `relay1`

Returns :

a string that identifies the sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

sensor→get_hardwareId()**YSensor****sensor→hardwareId()****sensor→get_hardwareId()**

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

sensor→**get_highestValue()**

YSensor

sensor→**highestValue()****sensor**→

get_highestValue()

Returns the maximal value observed for the measure since the device was started.

`double get_highestValue()`

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

sensor→**get_logFrequency()**
sensor→**logFrequency()****sensor**→
get_logFrequency()

YSensor

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

sensor→**get_logicalName()**

YSensor

sensor→**logicalName()****sensor**→
get_logicalName()

Returns the logical name of the sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

sensor→**get_lowestValue()****YSensor****sensor**→**lowestValue()****sensor**→
get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

sensor→get_module()

YSensor

sensor→module()**sensor→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

sensor→**get_recordedData()**
sensor→**recordedData()****sensor**→
get_recordedData()

YSensor

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

`YDataSet` **get_recordedData**(`s64` **startTime**, `s64` **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

sensor→**get_reportFrequency()****YSensor****sensor**→**reportFrequency()****sensor**→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()****Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

sensor→get_resolution()**YSensor****sensor→resolution()****sensor→get_resolution()**

Returns the resolution of the measured values.

`double get_resolution()`

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

sensor→**get_unit()**

YSensor

sensor→**unit()****sensor**→**get_unit()**

Returns the measuring unit for the measure.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

sensor→get_userData()**YSensor****sensor→userData()****sensor→get_userData()**

Returns the value of the userData attribute, as previously stored using method set_userData.

```
void * get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

sensor→**isOnline()****sensor**→**isOnline()**

YSensor

Checks if the sensor is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the sensor.

Returns :

`true` if the sensor can be reached, and `false` otherwise

sensor→load()**sensor→load()****YSensor**

Preloads the sensor cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

sensor→**loadCalibrationPoints()****sensor**→
loadCalibrationPoints()

YSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→nextSensor()**sensor→nextSensor()****YSensor**

Continues the enumeration of sensors started using `yFirstSensor()`.

`YSensor * nextSensor()`

Returns :

a pointer to a `YSensor` object, corresponding to a sensor currently online, or a `null` pointer if there are no more sensors to enumerate.

sensor→**registerTimedReportCallback()****sensor**→
registerTimedReportCallback()

YSensor

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YSensorTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

sensor→**registerValueCallback()****sensor**→
registerValueCallback()

YSensor

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YSensorValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

sensor→**set_highestValue()****YSensor****sensor**→**setHighestValue()****sensor**→**set_highestValue()**

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_logFrequency()****YSensor****sensor**→**setLogFrequency()****sensor**→**set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_logicalName()****YSensor****sensor**→**setLogicalName()****sensor**→
set_logicalName()

Changes the logical name of the sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

sensor→**set_lowestValue()****YSensor****sensor**→**setLowestValue()****sensor**→**set_lowestValue()**

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_reportFrequency()****YSensor****sensor**→**setReportFrequency()****sensor**→**set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_resolution()****YSensor****sensor**→**setResolution()****sensor**→**set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_userdata()

YSensor

sensor→setUserData()**sensor→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.37. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_servo.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YServo = yoctolib.YServo;
php	require_once('yocto_servo.php');
c++	#include "yocto_servo.h"
m	#import "yocto_servo.h"
pas	uses yocto_servo;
vb	yocto_servo.vb
cs	yocto_servo.cs
java	import com.yoctopuce.YoctoAPI.YServo;
py	from yocto_servo import *

Global functions

yFindServo(func)

Retrieves a servo for a given identifier.

yFirstServo()

Starts the enumeration of servos currently accessible.

YServo methods

servo→describe()

Returns a short text that describes unambiguously the instance of the servo in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

servo→get_advertisedValue()

Returns the current value of the servo (no more than 6 characters).

servo→get_enabled()

Returns the state of the servos.

servo→get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

servo→get_errorMessage()

Returns the error message of the latest error with the servo.

servo→get_errorType()

Returns the numerical error code of the latest error with the servo.

servo→get_friendlyName()

Returns a global identifier of the servo in the format `MODULE_NAME . FUNCTION_NAME`.

servo→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

servo→get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

servo→get_hardwareId()

Returns the unique hardware identifier of the servo in the form `SERIAL . FUNCTIONID`.

servo→get_logicalName()

Returns the logical name of the servo.

servo→get_module()

Gets the YModule object for the device on which the function is located.

servo→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_positionAtPowerOn()

Returns the servo position at device power up.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

servo→isOnline()

Checks if the servo is currently reachable, without raising any error.

servo→isOnline_async(callback, context)

Checks if the servo is currently reachable, without raising any error (asynchronous version).

servo→load(msValidity)

Preloads the servo cache with a specified validity duration.

servo→load_async(msValidity, callback, context)

Preloads the servo cache with a specified validity duration (asynchronous version).

servo→move(target, ms_duration)

Performs a smooth move at constant speed toward a given position.

servo→nextServo()

Continues the enumeration of servos started using yFirstServo().

servo→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

servo→set_enabled(newval)

Stops or starts the servo.

servo→set_enabledAtPowerOn(newval)

Configure the servo signal generator state at power up.

servo→set_logicalName(newval)

Changes the logical name of the servo.

servo→set_neutral(newval)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→set_position(newval)

Changes immediately the servo driving position.

servo→set_positionAtPowerOn(newval)

Configure the servo position at device power up.

servo→set_range(newval)

Changes the range of use of the servo, specified in per cents.

servo→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

servo→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YServo.FindServo()**YServo****yFindServo()****yFindServo()**

Retrieves a servo for a given identifier.

```
YServo* yFindServo( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the servo is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YServo.isOnline()` to test if the servo is indeed online at a given time. In case of ambiguity when looking for a servo by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the servo

Returns :

a `YServo` object allowing you to drive the servo.

YServo.FirstServo()
yFirstServo()**yFirstServo()**

YServo

Starts the enumeration of servos currently accessible.

`YServo* yFirstServo()`

Use the method `YServo.nextServo()` to iterate on next servos.

Returns :

a pointer to a `YServo` object, corresponding to the first servo currently online, or a `null` pointer if there are none.

servo→describe()**servo→describe()****YServo**

Returns a short text that describes unambiguously the instance of the servo in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the servo (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

servo→**get_advertisedValue()**
servo→**advertisedValue()****servo**→
get_advertisedValue()

YServo

Returns the current value of the servo (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the servo (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

servo→get_enabled()

YServo

servo→enabled()**servo→get_enabled()**

Returns the state of the servos.

`Y_ENABLED_enum` **get_enabled()**

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the state of the servos

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

servo→get_enabledAtPowerOn()**YServo****servo→enabledAtPowerOn()****servo→****get_enabledAtPowerOn()**

Returns the servo signal generator state at power up.

[Y_ENABLEDATPOWERON_enum](#) **get_enabledAtPowerOn()**

Returns :

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the servo signal generator state at power up

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

servo→get_errorMessage()

YServo

servo→errorMessage()**servo→**

get_errorMessage()

Returns the error message of the latest error with the servo.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the servo object

servo→get_errorType()**YServo****servo→errorType()****servo→get_errorType()**

Returns the numerical error code of the latest error with the servo.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the servo object

servo→**get_friendlyName()****YServo****servo**→**friendlyName()****servo**→**get_friendlyName()**

Returns a global identifier of the servo in the format `MODULE_NAME . FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the servo if they are defined, otherwise the serial number of the module and the hardware identifier of the servo (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the servo using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

servo→**get_functionDescriptor()**
servo→**functionDescriptor()****servo**→
get_functionDescriptor()

YServo

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

servo→get_functionId()

YServo

servo→functionId()**servo→get_functionId()**

Returns the hardware identifier of the servo, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the servo (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

servo→get_hardwareId()**YServo****servo→hardwareId()****servo→get_hardwareId()**

Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the servo. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the servo (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

servo→get_logicalName()

YServo

servo→logicalName()**servo→get_logicalName()**

Returns the logical name of the servo.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the servo. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

servo→get_module()**YServo****servo→module()****servo→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()** ()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

servo→get_neutral()

YServo

servo→neutral()**servo→get_neutral()**

Returns the duration in microseconds of a neutral pulse for the servo.

int **get_neutral()**

Returns :

an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns Y_NEUTRAL_INVALID.

servo→get_position()**YServo****servo→position()****servo→get_position()**

Returns the current servo position.

int **get_position**()

Returns :

an integer corresponding to the current servo position

On failure, throws an exception or returns Y_POSITION_INVALID.

servo→get_positionAtPowerOn()
servo→positionAtPowerOn()**servo→**
get_positionAtPowerOn()

YServo

Returns the servo position at device power up.

int get_positionAtPowerOn()

Returns :

an integer corresponding to the servo position at device power up

On failure, throws an exception or returns Y_POSITIONATPOWERON_INVALID.

servo→get_range()**YServo****servo→range()****servo→get_range()**

Returns the current range of use of the servo.

int **get_range()**

Returns :

an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns Y_RANGE_INVALID.

servo→get_userdata()

YServo

servo→userData()**servo→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

servo→isOnline()**servo→isOnline()****YServo**

Checks if the servo is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

Returns :

`true` if the servo can be reached, and `false` otherwise

servo→load()**servo→load()****YServo**

Preloads the servo cache with a specified validity duration.

YRETCODE **load(int msValidity)**

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

servo→move()**servo→move ()****YServo**

Performs a smooth move at constant speed toward a given position.

```
int move( int target, int ms_duration)
```

Parameters :

target new position at the end of the move
ms_duration total duration of the move, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→nextServo()**servo→nextServo()**

YServo

Continues the enumeration of servos started using `yFirstServo()`.

`YServo * nextServo()`

Returns :

a pointer to a `YServo` object, corresponding to a servo currently online, or a `null` pointer if there are no more servos to enumerate.

servo→registerValueCallback()**servo→**
registerValueCallback()

YServo

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YServoValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

servo→**set_enabled()****YServo****servo**→**setEnabled()****servo**→**set_enabled()**

Stops or starts the servo.

```
int set_enabled( Y_ENABLED_enum newval)
```

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_enabledAtPowerOn()**YServo****servo→setEnabledAtPowerOn()****servo→****set_enabledAtPowerOn()**

Configure the servo signal generator state at power up.

```
int set_enabledAtPowerOn( Y_ENABLEDATPOWERON_enum newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→**set_logicalName()****YServo****servo**→**setLogicalName()****servo**→
set_logicalName()

Changes the logical name of the servo.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the servo.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

servo→set_neutral()**YServo****servo→setNeutral()****servo→set_neutral()**

Changes the duration of the pulse corresponding to the neutral position of the servo.

```
int set_neutral( int newval)
```

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_position()

YServo

servo→setPosition()**servo→set_position()**

Changes immediately the servo driving position.

```
int set_position( int newval)
```

Parameters :

newval an integer corresponding to immediately the servo driving position

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_positionAtPowerOn()**YServo****servo→setPositionAtPowerOn()**
servo→set_positionAtPowerOn()

Configure the servo position at device power up.

```
int set_positionAtPowerOn( int newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_range()**YServo****servo→setRange()****servo→set_range()**

Changes the range of use of the servo, specified in per cents.

```
int set_range( int newval)
```

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms], When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the range of use of the servo, specified in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_userdata()**YServo****servo→setUserData()****servo→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.38. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_temperature.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YTemperature = yoctolib.YTemperature;
php	require_once('yocto_temperature.php');
c++	#include "yocto_temperature.h"
m	#import "yocto_temperature.h"
pas	uses yocto_temperature;
vb	yocto_temperature.vb
cs	yocto_temperature.cs
java	import com.yoctopuce.YoctoAPI.YTemperature;
py	from yocto_temperature import *

Global functions

yFindTemperature(func)

Retrieves a temperature sensor for a given identifier.

yFirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

YTemperature methods

temperature→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

temperature→describe()

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

temperature→get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

temperature→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

temperature→get_currentValue()

Returns the current value of the temperature.

temperature→get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

temperature→get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

temperature→get_friendlyName()

Returns a global identifier of the temperature sensor in the format `MODULE_NAME . FUNCTION_NAME`.

temperature→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

temperature→get_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

temperature→get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL . FUNCTIONID`.

temperature→get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

temperature→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

temperature→get_logicalName()

Returns the logical name of the temperature sensor.

temperature→get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

temperature→get_module()

Gets the `YModule` object for the device on which the function is located.

temperature→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

temperature→get_recordedData(startTime, endTime)

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

temperature→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

temperature→get_resolution()

Returns the resolution of the measured values.

temperature→get_sensorType()

Returns the temperature sensor type.

temperature→get_unit()

Returns the measuring unit for the temperature.

temperature→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

temperature→isOnline()

Checks if the temperature sensor is currently reachable, without raising any error.

temperature→isOnline_async(callback, context)

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

temperature→load(msValidity)

Preloads the temperature sensor cache with a specified validity duration.

temperature→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

temperature→load_async(msValidity, callback, context)

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

temperature→nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

temperature→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

temperature→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

temperature→set_highestValue(newval)

Changes the recorded maximal value observed.

temperature→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

3. Reference

temperature→set_logicalName(newval)

Changes the logical name of the temperature sensor.

temperature→set_lowestValue(newval)

Changes the recorded minimal value observed.

temperature→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

temperature→set_resolution(newval)

Changes the resolution of the measured physical values.

temperature→set_sensorType(newval)

Modify the temperature sensor type.

temperature→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

temperature→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTemperature.FindTemperature() yFindTemperature()yFindTemperature()

YTemperature

Retrieves a temperature sensor for a given identifier.

```
YTemperature* yFindTemperature( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the temperature sensor

Returns :

a `YTemperature` object allowing you to drive the temperature sensor.

YTemperature.FirstTemperature()

YTemperature

yFirstTemperature()**yFirstTemperature()**

Starts the enumeration of temperature sensors currently accessible.

YTemperature* **yFirstTemperature()**

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()temperature→
calibrateFromPoints()

YTemperature

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→describe()**YTemperature**

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the temperature sensor (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

temperature→**get_advertisedValue()****YTemperature****temperature**→**advertisedValue()****temperature**→
get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the temperature sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

temperature→get_currentRawValue()

YTemperature

temperature→currentRawValue()temperature→
get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

temperature→**get_currentValue()****YTemperature****temperature**→**currentValue()****temperature**→
get_currentValue()

Returns the current value of the temperature.

double **get_currentValue()**

Returns :

a floating point number corresponding to the current value of the temperature

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

temperature→get_errorMessage()

YTemperature

temperature→errorMessage()temperature→

get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the temperature sensor object

temperature→**get_errorType()****YTemperature****temperature**→**errorType()****temperature**→
get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the temperature sensor object

temperature→get_friendlyName()

YTemperature

temperature→friendlyName()**temperature→**

get_friendlyName()

Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the temperature sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the temperature sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the temperature sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

temperature→**get_functionDescriptor()****YTemperature****temperature**→**functionDescriptor()****temperature**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

temperature→**get_functionId()**

YTemperature

temperature→**functionId()****temperature**→
get_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the temperature sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

temperature→**get_hardwareId()****YTemperature****temperature**→**hardwareId()****temperature**→
get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the temperature sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the temperature sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

temperature→**get_highestValue()** **YTemperature**
temperature→**highestValue()****temperature**→
get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

`double get_highestValue()`

Returns :

a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

temperature→**get_logFrequency()****YTemperature****temperature**→**logFrequency()****temperature**→
get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
string get_logFrequency( )
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

temperature→**get_logicalName()**

YTemperature

temperature→**logicalName()****temperature**→
get_logicalName()

Returns the logical name of the temperature sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the temperature sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

temperature→**get_lowestValue()****YTemperature****temperature**→**lowestValue()****temperature**→
get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

`double` **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

temperature→get_module()

YTemperature

temperature→module()**temperature→**
get_module()

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

temperature→**get_recordedData()****YTemperature****temperature**→**recordedData()****temperature**→
get_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YDataSet **get_recordedData**(s64 **startTime**, s64 **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

temperature→get_reportFrequency()

YTemperature

temperature→reportFrequency()temperature→

get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string get_reportFrequency()

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

temperature→**get_resolution()****YTemperature****temperature**→**resolution()****temperature**→
get_resolution()

Returns the resolution of the measured values.

`double` **get_resolution()** `()`

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

temperature→**get_sensorType()****YTemperature****temperature**→**sensorType()****temperature**→
get_sensorType()

Returns the temperature sensor type.

[Y_SENSORTYPE_enum](#) **get_sensorType()**

Returns :

a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES corresponding to the temperature sensor type

On failure, throws an exception or returns Y_SENSORTYPE_INVALID.

temperature→**get_unit()****YTemperature****temperature**→**unit()****temperature**→**get_unit()**

Returns the measuring unit for the temperature.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the temperature

On failure, throws an exception or returns Y_UNIT_INVALID.

temperature→get_userdata()

YTemperature

temperature→userdata()**temperature→**
get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

temperature→isOnline()**temperature→isOnline()****YTemperature**

Checks if the temperature sensor is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

Returns :

`true` if the temperature sensor can be reached, and `false` otherwise

temperature→load()**temperature→load()****YTemperature**

Preloads the temperature sensor cache with a specified validity duration.

YRETCODE load(int msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

temperature→**loadCalibrationPoints()****temperature**→
loadCalibrationPoints()

YTemperature

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**nextTemperature()****temperature**→
nextTemperature()

YTemperature

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

`YTemperature * nextTemperature()`

Returns :

a pointer to a `YTemperature` object, corresponding to a temperature sensor currently online, or a `null` pointer if there are no more temperature sensors to enumerate.

temperature→registerTimedReportCallback()**YTemperature****temperature→registerTimedReportCallback()**

Registers the callback function that is invoked on every periodic timed notification.

int **registerTimedReportCallback**(YTemperatureTimedReportCallback **callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

temperature → registerValueCallback() temperature	YTemperature
→ registerValueCallback()	

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YTemperatureValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

temperature→**set_highestValue()****YTemperature****temperature**→**setHighestValue()****temperature**→
set_highestValue()

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_logFrequency()**

YTemperature

temperature→**setLogFrequency()****temperature**→

set_logFrequency()

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_logicalName()****YTemperature****temperature**→**setLogicalName()****temperature**→
set_logicalName()

Changes the logical name of the temperature sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the temperature sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

temperature→**set_lowestValue()**

YTemperature

temperature→**setLowestValue()****temperature**→
set_lowestValue()

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_reportFrequency()**YTemperature****temperature→setReportFrequency()**
temperature→set_reportFrequency()

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_resolution()**

YTemperature

temperature→**setResolution()****temperature**→
set_resolution()

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_sensorType()**

YTemperature

temperature→**setSensorType()****temperature**→
set_sensorType()

Modify the temperature sensor type.

```
int set_sensorType( Y_SENSORTYPE_enum newval)
```

This function is used to to define the type of thermocouple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_userdata()

YTemperature

temperature→setUserData()
temperature→set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.39. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_tilt.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YTilt = yoctolib.YTilt;</code>
php	<code>require_once('yocto_tilt.php');</code>
c++	<code>#include "yocto_tilt.h"</code>
m	<code>#import "yocto_tilt.h"</code>
pas	<code>uses yocto_tilt;</code>
vb	<code>yocto_tilt.vb</code>
cs	<code>yocto_tilt.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YTilt;</code>
py	<code>from yocto_tilt import *</code>

Global functions

yFindTilt(func)

Retrieves a tilt sensor for a given identifier.

yFirstTilt()

Starts the enumeration of tilt sensors currently accessible.

YTilt methods

tilt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

tilt→describe()

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

tilt→get_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

tilt→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

tilt→get_currentValue()

Returns the current value of the inclination.

tilt→get_errorMessage()

Returns the error message of the latest error with the tilt sensor.

tilt→get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

tilt→get_friendlyName()

Returns a global identifier of the tilt sensor in the format `MODULE_NAME . FUNCTION_NAME`.

tilt→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

tilt→get_functionId()

Returns the hardware identifier of the tilt sensor, without reference to the module.

tilt→get_hardwareId()

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL . FUNCTIONID`.

tilt→get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

tilt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

tilt→get_logicalName()

Returns the logical name of the tilt sensor.

tilt→get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

tilt→get_module()

Gets the YModule object for the device on which the function is located.

tilt→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

tilt→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

tilt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

tilt→get_resolution()

Returns the resolution of the measured values.

tilt→get_unit()

Returns the measuring unit for the inclination.

tilt→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

tilt→isOnline()

Checks if the tilt sensor is currently reachable, without raising any error.

tilt→isOnline_async(callback, context)

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

tilt→load(msValidity)

Preloads the tilt sensor cache with a specified validity duration.

tilt→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

tilt→load_async(msValidity, callback, context)

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

tilt→nextTilt()

Continues the enumeration of tilt sensors started using yFirstTilt().

tilt→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

tilt→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

tilt→set_highestValue(newval)

Changes the recorded maximal value observed.

tilt→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

tilt→set_logicalName(newval)

Changes the logical name of the tilt sensor.

tilt→**set_lowestValue**(newval)

Changes the recorded minimal value observed.

tilt→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

tilt→**set_resolution**(newval)

Changes the resolution of the measured physical values.

tilt→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

tilt→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTilt.FindTilt()**YTilt****yFindTilt()****yFindTilt()**

Retrieves a tilt sensor for a given identifier.

```
YTilt* yFindTilt( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the tilt sensor

Returns :

a `YTilt` object allowing you to drive the tilt sensor.

YTilt.FirstTilt()**YTilt****yFirstTilt()****yFirstTilt()**

Starts the enumeration of tilt sensors currently accessible.

YTilt* yFirstTilt()

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

Returns :

a pointer to a `YTilt` object, corresponding to the first tilt sensor currently online, or a `null` pointer if there are none.

tilt→calibrateFromPoints()**tilt→**
calibrateFromPoints()

YTilt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→describe()**tilt→describe()****YTilt**

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the tilt sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

tilt→get_advertisedValue()

YTilt

tilt→advertisedValue()**tilt→**

get_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the tilt sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

tilt→**get_currentRawValue()****YTilt****tilt**→**currentRawValue()****tilt**→**get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

tilt→get_currentValue()

YTilt

tilt→currentValue()**tilt→get_currentValue()**

Returns the current value of the inclination.

double **get_currentValue()**

Returns :

a floating point number corresponding to the current value of the inclination

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

tilt→get_errorMessage()**YTilt****tilt→errorMessage()****tilt→get_errorMessage()**

Returns the error message of the latest error with the tilt sensor.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the tilt sensor object

tilt→get_errorType()

YTilt

tilt→errorType()**tilt→get_errorType()**

Returns the numerical error code of the latest error with the tilt sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the tilt sensor object

tilt→get_friendlyName()**YTilt****tilt→friendlyName()****tilt→get_friendlyName()**

Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the tilt sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the tilt sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the tilt sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

tilt→get_functionDescriptor()
tilt→functionDescriptor()
get_functionDescriptor()

YTilt

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

tilt→get_functionId()**YTilt****tilt→functionId()****tilt→get_functionId()**

Returns the hardware identifier of the tilt sensor, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the tilt sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

tilt→**get_hardwareId()****YTilt****tilt**→**hardwareId()****tilt**→**get_hardwareId()**

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the tilt sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the tilt sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

tilt→get_highestValue()**YTilt****tilt→highestValue()****tilt→get_highestValue()**

Returns the maximal value observed for the inclination since the device was started.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the inclination since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

tilt→get_logFrequency()**YTilt****tilt→logFrequency()****tilt→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

tilt→get_logicalName()**YTilt****tilt→logicalName()****tilt→get_logicalName()**

Returns the logical name of the tilt sensor.

```
string get_logicalName( )
```

Returns :

a string corresponding to the logical name of the tilt sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

tilt→get_lowestValue()

YTilt

tilt→lowestValue()**tilt→get_lowestValue()**

Returns the minimal value observed for the inclination since the device was started.

double **get_lowestValue()** ()

Returns :

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

tilt→get_module()**YTilt****tilt→module()****tilt→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

tilt→get_recordedData()**YTilt****tilt→recordedData()****tilt→get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YDataSet get_recordedData( s64 startTime, s64 endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

tilt→get_reportFrequency()**YTilt****tilt→reportFrequency()****tilt→****get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()****Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

tilt→get_resolution()

YTilt

tilt→resolution()**tilt→get_resolution()**

Returns the resolution of the measured values.

double **get_resolution()**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

tilt→get_unit()**YTilt****tilt→unit()****tilt→get_unit()**

Returns the measuring unit for the inclination.

string **get_unit**()

Returns :

a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns Y_UNIT_INVALID.

tilt→get_userdata()

YTilt

tilt→userdata()**tilt→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

tilt→isOnline()**tilt→isOnline()****YTilt**

Checks if the tilt sensor is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

Returns :

`true` if the tilt sensor can be reached, and `false` otherwise

tilt→load()**tilt→load()****YTilt**

Preloads the tilt sensor cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

tilt→loadCalibrationPoints()
loadCalibrationPoints()

YTilt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→nextTilt()`tilt→nextTilt()`

YTilt

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

`YTilt * nextTilt()`

Returns :

a pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a `null` pointer if there are no more tilt sensors to enumerate.

tilt→registerTimedReportCallback()**tilt→**
registerTimedReportCallback()

YTilt

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YTiltTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

tilt→registerValueCallback()**tilt→**
registerValueCallback()

YTilt

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YTiltValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

tilt→set_highestValue()**YTilt****tilt→setHighestValue()****tilt→set_highestValue()**

Changes the recorded maximal value observed.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logFrequency()
tilt→setLogFrequency()
set_logFrequency()

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logicalName()**YTilt****tilt→setLogicalName()****tilt→set_logicalName()**

Changes the logical name of the tilt sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the tilt sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

tilt→set_lowestValue()**YTilt****tilt→setLowestValue()****tilt→set_lowestValue()**

Changes the recorded minimal value observed.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_reportFrequency()**YTilt****tilt→setReportFrequency()****tilt→****set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_resolution()**YTilt****tilt→setResolution()****tilt→set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_userdata()**YTilt****tilt→setUserData()****tilt→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.40. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_voc.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YVoc = yoctolib.YVoc;</code>
php	<code>require_once('yocto_voc.php');</code>
c++	<code>#include "yocto_voc.h"</code>
m	<code>#import "yocto_voc.h"</code>
pas	<code>uses yocto_voc;</code>
vb	<code>yocto_voc.vb</code>
cs	<code>yocto_voc.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YVoc;</code>
py	<code>from yocto_voc import *</code>

Global functions

yFindVoc(func)

Retrieves a Volatile Organic Compound sensor for a given identifier.

yFirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

YVoc methods

voc→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voc→describe()

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

voc→get_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

voc→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

voc→get_currentValue()

Returns the current measure for the estimated VOC concentration.

voc→get_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

voc→get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

voc→get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME . FUNCTION_NAME`.

voc→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voc→get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

voc→get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

`voc→get_highestValue()`

Returns the maximal value observed for the estimated VOC concentration.

`voc→get_logFrequency()`

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

`voc→get_logicalName()`

Returns the logical name of the Volatile Organic Compound sensor.

`voc→get_lowestValue()`

Returns the minimal value observed for the estimated VOC concentration.

`voc→get_module()`

Gets the `YModule` object for the device on which the function is located.

`voc→get_module_async(callback, context)`

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`voc→get_recordedData(startTime, endTime)`

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

`voc→get_reportFrequency()`

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

`voc→get_resolution()`

Returns the resolution of the measured values.

`voc→get_unit()`

Returns the measuring unit for the estimated VOC concentration.

`voc→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

`voc→isOnline()`

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

`voc→isOnline_async(callback, context)`

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

`voc→load(msValidity)`

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

`voc→loadCalibrationPoints(rawValues, refValues)`

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

`voc→load_async(msValidity, callback, context)`

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

`voc→nextVoc()`

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

`voc→registerTimedReportCallback(callback)`

Registers the callback function that is invoked on every periodic timed notification.

`voc→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`voc→set_highestValue(newval)`

Changes the recorded maximal value observed for the estimated VOC concentration.

3. Reference

voc→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voc→set_logicalName(newval)

Changes the logical name of the Volatile Organic Compound sensor.

voc→set_lowestValue(newval)

Changes the recorded minimal value observed for the estimated VOC concentration.

voc→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voc→set_resolution(newval)

Changes the resolution of the measured physical values.

voc→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voc→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoc.FindVoc()**YVoc****yFindVoc()****yFindVoc()**

Retrieves a Volatile Organic Compound sensor for a given identifier.

```
YVoc* yFindVoc( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Volatile Organic Compound sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoc.isOnline()` to test if the Volatile Organic Compound sensor is indeed online at a given time. In case of ambiguity when looking for a Volatile Organic Compound sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the Volatile Organic Compound sensor

Returns :

a `YVoc` object allowing you to drive the Volatile Organic Compound sensor.

YVoc.FirstVoc()**YVoc****yFirstVoc()****yFirstVoc()**

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

YVoc* yFirstVoc()

Use the method `YVoc.nextVoc()` to iterate on next Volatile Organic Compound sensors.

Returns :

a pointer to a `YVoc` object, corresponding to the first Volatile Organic Compound sensor currently online, or a `null` pointer if there are none.

voc→calibrateFromPoints()**voc→**
calibrateFromPoints()

YVoc

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→describe()**voc→describe()****YVoc**

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the Volatile Organic Compound sensor (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voc→**get_advertisedValue()****YVoc****voc**→**advertisedValue()****voc**→**get_advertisedValue()**

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

voc→**get_currentRawValue()**

YVoc

voc→**currentRawValue()****voc**→

get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

voc→get_currentValue()**YVoc****voc→currentValue()****voc→get_currentValue()**

Returns the current measure for the estimated VOC concentration.

`double get_currentValue()`

Returns :

a floating point number corresponding to the current measure for the estimated VOC concentration

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

voc→**get_errorMessage()**

YVoc

voc→**errorMessage()****voc**→**get_errorMessage()**

Returns the error message of the latest error with the Volatile Organic Compound sensor.

`string` **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the Volatile Organic Compound sensor object

`voc→get_errorType()`**YVoc****`voc→errorType()`****`voc→get_errorType()`**

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

`YRETCODE` `get_errorType()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the Volatile Organic Compound sensor object

voc→**get_friendlyName()****YVoc****voc**→**friendlyName()****voc**→**get_friendlyName()**

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

string **get_friendlyName()**

The returned string uses the logical names of the module and of the Volatile Organic Compound sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the Volatile Organic Compound sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

YVoc
YVoc→**get_functionDescriptor()**
YVoc→**functionDescriptor()**
YVoc→**get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voc→**get_functionId()**

YVoc

voc→**functionId()****voc**→**get_functionId()**

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

string **get_functionId()** ()

For example `relay1`

Returns :

a string that identifies the Volatile Organic Compound sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voc→get_hardwareId()**YVoc****voc→hardwareId()****voc→get_hardwareId()**

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

`string get_hardwareId()`

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Volatile Organic Compound sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the Volatile Organic Compound sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

voc→**get_highestValue()**

YVoc

voc→**highestValue()****voc**→**get_highestValue()**

Returns the maximal value observed for the estimated VOC concentration.

double **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the estimated VOC concentration

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

voc→get_logFrequency()**YVoc****voc→logFrequency()****voc→get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string **get_logFrequency()**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

voc→get_logicalName()

YVoc

voc→logicalName()**voc→get_logicalName()**

Returns the logical name of the Volatile Organic Compound sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the Volatile Organic Compound sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

voc→get_lowestValue()**YVoc****voc→lowestValue()****voc→get_lowestValue()**

Returns the minimal value observed for the estimated VOC concentration.

double **get_lowestValue()**

Returns :

a floating point number corresponding to the minimal value observed for the estimated VOC concentration

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

voc→get_module()

YVoc

voc→module()**voc→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

voc→**get_recordedData()****YVoc****voc**→**recordedData()****voc**→**get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YDataSet get_recordedData( s64 startTime, s64 endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voc→**get_reportFrequency()****YVoc****voc**→**reportFrequency()****voc**→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

string **get_reportFrequency()****Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

voc→get_resolution()**YVoc****voc→resolution()****voc→get_resolution()**

Returns the resolution of the measured values.

double **get_resolution()** ()

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

voc→get_unit()

YVoc

voc→unit()**voc→get_unit()**

Returns the measuring unit for the estimated VOC concentration.

string **get_unit()**

Returns :

a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns Y_UNIT_INVALID.

voc→get_userdata()**YVoc****voc→userdata()****voc→get_userdata()**

Returns the value of the userData attribute, as previously stored using method set_userdata.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

voc→**isOnline()****voc**→**isOnline()****YVoc**

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Volatile Organic Compound sensor.

Returns :

`true` if the Volatile Organic Compound sensor can be reached, and `false` otherwise

voc→load()**voc→load()****YVoc**

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

voc→**loadCalibrationPoints()****voc**→
loadCalibrationPoints()

YVoc

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

`voc→nextVoc()`**`voc→nextVoc()`****YVoc**

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

`YVoc * nextVoc()`

Returns :

a pointer to a `YVoc` object, corresponding to a Volatile Organic Compound sensor currently online, or a `null` pointer if there are no more Volatile Organic Compound sensors to enumerate.

voc→**registerTimedReportCallback()****voc**→
registerTimedReportCallback()

YVoc

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YVocTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

YVoc→registerValueCallback()**YVoc→**
registerValueCallback()

YVoc

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YVocValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voc→**set_highestValue()**

YVoc

voc→**setHighestValue()****voc**→**set_highestValue()**

Changes the recorded maximal value observed for the estimated VOC concentration.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the estimated VOC concentration

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→**set_logFrequency()****YVoc****voc**→**setLogFrequency()****voc**→**set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→**set_logicalName()****YVoc****voc**→**setLogicalName()****voc**→**set_logicalName()**

Changes the logical name of the Volatile Organic Compound sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Volatile Organic Compound sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

voc→**set_lowestValue()****YVoc****voc**→**setLowestValue()****voc**→**set_lowestValue()**

Changes the recorded minimal value observed for the estimated VOC concentration.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the estimated VOC concentration

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

newval a string corresponding to the timed value notification frequency for this function

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_resolution()**YVoc****voc→setResolution()****voc→set_resolution()**

Changes the resolution of the measured physical values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→**set_userData()**

YVoc

voc→**setUserData()****voc**→**set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userData( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.41. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voltage.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YVoltage = yoctolib.YVoltage;
php	require_once('yocto_voltage.php');
c++	#include "yocto_voltage.h"
m	#import "yocto_voltage.h"
pas	uses yocto_voltage;
vb	yocto_voltage.vb
cs	yocto_voltage.cs
java	import com.yoctopuce.YoctoAPI.YVoltage;
py	from yocto_voltage import *

Global functions

yFindVoltage(func)

Retrieves a voltage sensor for a given identifier.

yFirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

YVoltage methods

voltage→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voltage→describe()

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

voltage→get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

voltage→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

voltage→get_currentValue()

Returns the current measure for the voltage.

voltage→get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

voltage→get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

voltage→get_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME . FUNCTION_NAME`.

voltage→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voltage→get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

voltage→get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL . FUNCTIONID`.

voltage→get_highestValue()

Returns the maximal value observed for the voltage.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage.

voltage→get_module()

Gets the YModule object for the device on which the function is located.

voltage→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(callback, context)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→load(msValidity)

Preloads the voltage sensor cache with a specified validity duration.

voltage→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

voltage→load_async(msValidity, callback, context)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using yFirstVoltage().

voltage→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_highestValue(newval)

Changes the recorded maximal value observed pour the voltage.

voltage→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(newval)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed pour the voltage.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured values.

voltage→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltage.FindVoltage() yFindVoltage()yFindVoltage()

YVoltage

Retrieves a voltage sensor for a given identifier.

```
YVoltage* yFindVoltage( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FirstVoltage()**YVoltage****yFirstVoltage()****yFirstVoltage()**

Starts the enumeration of voltage sensors currently accessible.

YVoltage* **yFirstVoltage()**

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()**voltage→**
calibrateFromPoints()

YVoltage

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
int calibrateFromPoints( vector<double> rawValues,  
                        vector<double> refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→describe()**voltage→describe()****YVoltage**

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voltage→get_advertisedValue()

YVoltage

voltage→advertisedValue()**voltage→**

get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

string get_advertisedValue()

Returns :

a string corresponding to the current value of the voltage sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

voltage→**get_currentRawValue()****YVoltage****voltage**→**currentRawValue()****voltage**→
get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

double **get_currentRawValue()**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

voltage→get_currentValue()

YVoltage

voltage→currentValue()

get_currentValue()

Returns the current measure for the voltage.

`double get_currentValue()`

Returns :

a floating point number corresponding to the current measure for the voltage

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

voltage→**get_errorMessage()****YVoltage****voltage**→**errorMessage()****voltage**→
get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

```
string get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the voltage sensor object

voltage→get_errorType()

YVoltage

voltage→errorType()**voltage→get_errorType()**

Returns the numerical error code of the latest error with the voltage sensor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the voltage sensor object

voltage→**get_friendlyName()****YVoltage****voltage**→**friendlyName()****voltage**→
get_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the voltage sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

voltage→**get_functionDescriptor()**

YVoltage

voltage→**functionDescriptor()****voltage**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voltage→get_functionId()**YVoltage****voltage→functionId()****voltage→get_functionId()**

Returns the hardware identifier of the voltage sensor, without reference to the module.

string **get_functionId()**

For example `relay1`

Returns :

a string that identifies the voltage sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

voltage→get_hardwareId()

YVoltage

voltage→hardwareId()**voltage→get_hardwareId()**

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the voltage sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

voltage→**get_highestValue()**
voltage→**highestValue()****voltage**→
get_highestValue()

YVoltage

Returns the maximal value observed for the voltage.

`double` **get_highestValue()**

Returns :

a floating point number corresponding to the maximal value observed for the voltage

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

voltage→get_logFrequency()

YVoltage

voltage→logFrequency()**voltage→**
get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

string get_logFrequency()

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

voltage→**get_logicalName()****YVoltage****voltage**→**logicalName()****voltage**→
get_logicalName()

Returns the logical name of the voltage sensor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the voltage sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

voltage→**get_lowestValue()**

YVoltage

voltage→**lowestValue()****voltage**→
get_lowestValue()

Returns the minimal value observed for the voltage.

`double get_lowestValue()`

Returns :

a floating point number corresponding to the minimal value observed for the voltage

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

voltage→get_module()**YVoltage****voltage→module()****voltage→get_module()**

Gets the YModule object for the device on which the function is located.

```
YModule * get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

voltage→**get_recordedData()****YVoltage****voltage**→**recordedData()****voltage**→**get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

`YDataSet` **get_recordedData**(`s64` **startTime**, `s64` **endTime**)

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voltage→get_reportFrequency()**YVoltage****voltage→reportFrequency()****voltage→****get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
string get_reportFrequency( )
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

voltage→get_resolution()

YVoltage

voltage→resolution()**voltage→get_resolution()**

Returns the resolution of the measured values.

double **get_resolution()** ()

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

voltage→**get_unit()****YVoltage****voltage**→**unit()****voltage**→**get_unit()**

Returns the measuring unit for the voltage.

string **get_unit()** ()

Returns :

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns Y_UNIT_INVALID.

voltage→get_userdata()

YVoltage

voltage→userdata()**voltage→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

voltage→isOnline()**voltage→isOnline()****YVoltage**

Checks if the voltage sensor is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

Returns :

`true` if the voltage sensor can be reached, and `false` otherwise

voltage→load()**voltage→load()****YVoltage**

Preloads the voltage sensor cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

voltage→loadCalibrationPoints()**voltage→**
loadCalibrationPoints()

YVoltage

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
int loadCalibrationPoints( vector<double>& rawValues,  
                           vector<double>& refValues)
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→nextVoltage()**voltage→nextVoltage()**

YVoltage

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

`YVoltage * nextVoltage()`

Returns :

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a `null` pointer if there are no more voltage sensors to enumerate.

voltage→**registerTimedReportCallback()****voltage**→
registerTimedReportCallback()

YVoltage

Registers the callback function that is invoked on every periodic timed notification.

```
int registerTimedReportCallback( YVoltageTimedReportCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

voltage→registerValueCallback()**voltage→**
registerValueCallback()

YVoltage

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YVoltageValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voltage→**set_highestValue()**
voltage→**setHighestValue()****voltage**→
set_highestValue()

YVoltage

Changes the recorded maximal value observed pour the voltage.

```
int set_highestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed pour the voltage

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_logFrequency()**

YVoltage

voltage→**setLogFrequency()****voltage**→

set_logFrequency()

Changes the datalogger recording frequency for this function.

```
int set_logFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_logicalName()****YVoltage****voltage**→**setLogicalName()****voltage**→
set_logicalName()

Changes the logical name of the voltage sensor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

voltage→**set_lowestValue()**

YVoltage

voltage→**setLowestValue()****voltage**→
set_lowestValue()

Changes the recorded minimal value observed pour the voltage.

```
int set_lowestValue( double newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed pour the voltage

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_reportFrequency()**YVoltage****voltage→setReportFrequency()****voltage→****set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
int set_reportFrequency( const string& newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_resolution()****YVoltage****voltage**→**setResolution()****voltage**→
set_resolution()

Changes the resolution of the measured values.

```
int set_resolution( double newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_userdata()**YVoltage****voltage→setUserData()****voltage→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.42. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_vsource.js'></script></code>
php	<code>require_once('yocto_vsource.php');</code>
c++	<code>#include "yocto_vsource.h"</code>
m	<code>#import "yocto_vsource.h"</code>
pas	<code>uses yocto_vsource;</code>
vb	<code>yocto_vsource.vb</code>
cs	<code>yocto_vsource.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YVSource;</code>
py	<code>from yocto_vsource import *</code>

Global functions	
yFindVSource(func)	Retrieves a voltage source for a given identifier.
yFirstVSource()	Starts the enumeration of voltage sources currently accessible.
YVSource methods	
vsource→describe()	Returns a short text that describes the function in the form <code>TYPE (NAME) = SERIAL . FUNCTIONID</code> .
vsource→get_advertisedValue()	Returns the current value of the voltage source (no more than 6 characters).
vsource→get_errorMessage()	Returns the error message of the latest error with this function.
vsource→get_errorType()	Returns the numerical error code of the latest error with this function.
vsource→get_extPowerFailure()	Returns true if external power supply voltage is too low.
vsource→get_failure()	Returns true if the module is in failure mode.
vsource→get_friendlyName()	Returns a global identifier of the function in the format <code>MODULE_NAME . FUNCTION_NAME</code> .
vsource→get_functionDescriptor()	Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
vsource→get_functionId()	Returns the hardware identifier of the function, without reference to the module.
vsource→get_hardwareId()	Returns the unique hardware identifier of the function in the form <code>SERIAL . FUNCTIONID</code> .
vsource→get_logicalName()	Returns the logical name of the voltage source.
vsource→get_module()	Gets the <code>YModule</code> object for the device on which the function is located.
vsource→get_module_async(callback, context)	

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`vsources→get_overCurrent()`

Returns true if the appliance connected to the device is too greedy .

`vsources→get_overHeat()`

Returns TRUE if the module is overheating.

`vsources→get_overLoad()`

Returns true if the device is not able to maintain the requested voltage output .

`vsources→get_regulationFailure()`

Returns true if the voltage output is too high regarding the requested voltage .

`vsources→get_unit()`

Returns the measuring unit for the voltage.

`vsources→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

`vsources→get_voltage()`

Returns the voltage output command (mV)

`vsources→isOnline()`

Checks if the function is currently reachable, without raising any error.

`vsources→isOnline_async(callback, context)`

Checks if the function is currently reachable, without raising any error (asynchronous version).

`vsources→load(msValidity)`

Preloads the function cache with a specified validity duration.

`vsources→load_async(msValidity, callback, context)`

Preloads the function cache with a specified validity duration (asynchronous version).

`vsources→nextVSource()`

Continues the enumeration of voltage sources started using `yFirstVSource()` .

`vsources→pulse(voltage, ms_duration)`

Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.

`vsources→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`vsources→set_logicalName(newval)`

Changes the logical name of the voltage source.

`vsources→set_userData(data)`

Stores a user context provided as argument in the `userData` attribute of the function.

`vsources→set_voltage(newval)`

Tunes the device output voltage (milliVolts).

`vsources→voltageMove(target, ms_duration)`

Performs a smooth move at constant speed toward a given value.

`vsources→wait_async(callback, context)`

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

yFindVSource() —**YVSource****YVSource.FindVSource()****yFindVSource()**

Retrieves a voltage source for a given identifier.

YVSource* **yFindVSource**(const string& **func**)**yFindVSource()** — **YVSource.FindVSource()****yFindVSource()**

Retrieves a voltage source for a given identifier.

js	function yFindVSource (func)
php	function yFindVSource (\$func)
cpp	YVSource* yFindVSource (const string& func)
m	YVSource* yFindVSource (NSString* func)
pas	function yFindVSource (func : string): TYVSource
vb	function yFindVSource (ByVal func As String) As YVSource
cs	YVSource FindVSource (string func)
java	YVSource FindVSource (String func)
py	def FindVSource (func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :**func** a string that uniquely characterizes the voltage source**Returns :**a **YVSource** object allowing you to drive the voltage source.

yFirstVSource() —**YVSource****YVSource.FirstVSource()****yFirstVSource()**

Starts the enumeration of voltage sources currently accessible.

YVSource* **yFirstVSource()**

yFirstVSource() — **YVSource.FirstVSource()****yFirstVSource()**

Starts the enumeration of voltage sources currently accessible.

js	function yFirstVSource()
php	function yFirstVSource()
cpp	YVSource* yFirstVSource()
m	YVSource* yFirstVSource()
pas	function yFirstVSource() : TYVSource
vb	function yFirstVSource() As YVSource
cs	YVSource FirstVSource()
java	YVSource FirstVSource()
py	def FirstVSource()

Use the method `YVSource.nextVSource()` to iterate on next voltage sources.

Returns :

a pointer to a `YVSource` object, corresponding to the first voltage source currently online, or a `null` pointer if there are none.

vsourcesource→describe()vsourcesource→describe()**YVSource**

Returns a short text that describes the function in the form `TYPE (NAME) =SERIAL.FUNCTIONID`.

`string describe()`

vsourcesource→describe()vsourcesource→describe()

Returns a short text that describes the function in the form `TYPE (NAME) =SERIAL.FUNCTIONID`.

js	function describe()
php	function describe()
cpp	string describe()
m	-(NSString*) describe
pas	function describe() : string
vb	function describe() As String
cs	string describe()
java	String describe()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the function (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

vsource→**get_advertisedValue()****YVSource****vsource**→**advertisedValue()****vsource**→**get_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

string **get_advertisedValue()****vsource**→**get_advertisedValue()****vsource**→**advertisedValue()****vsource**→**get_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

js	function get_advertisedValue()
php	function get_advertisedValue()
cpp	string get_advertisedValue()
m	-(NSString*) advertisedValue
pas	function get_advertisedValue() : string
vb	function get_advertisedValue() As String
cs	string get_advertisedValue()
java	String get_advertisedValue()
py	def get_advertisedValue()
cmd	YVSource target get_advertisedValue

Returns :

a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

vsource→**get_errorMessage()****YVSource****vsource**→**errorMessage()****vsource**→**get_errorMessage()**

Returns the error message of the latest error with this function.

```
string get_errorMessage( )
```

vsource→**get_errorMessage()****vsource**→**errorMessage()****vsource**→**get_errorMessage()**

Returns the error message of the latest error with this function.

js	function get_errorMessage ()
php	function get_errorMessage ()
cpp	string get_errorMessage ()
m	-(NSString*) errorMessage
pas	function get_errorMessage (): string
vb	function get_errorMessage () As String
cs	string get_errorMessage ()
java	String get_errorMessage ()
py	def get_errorMessage ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this function object

vsource→**get_errorType()****YVSource****vsource**→**errorType()****vsource**→**get_errorType()**

Returns the numerical error code of the latest error with this function.

YRETCODE **get_errorType()**

vsource→**get_errorType()****vsource**→**errorType()****vsource**→**get_errorType()**

Returns the numerical error code of the latest error with this function.

<code>js</code>	<code>function get_errorType()</code>
<code>php</code>	<code>function get_errorType()</code>
<code>cpp</code>	<code>YRETCODE get_errorType()</code>
<code>pas</code>	<code>function get_errorType(): YRETCODE</code>
<code>vb</code>	<code>function get_errorType() As YRETCODE</code>
<code>cs</code>	<code>YRETCODE get_errorType()</code>
<code>java</code>	<code>int get_errorType()</code>
<code>py</code>	<code>def get_errorType()</code>

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

vsourcetoget_extPowerFailure()

vsourcetoextPowerFailure()vsourcetoget_extPowerFailure()

YVSource

Returns true if external power supply voltage is too low.

Y_EXTPOWERFAILURE_enum get_extPowerFailure()

vsourcetoget_extPowerFailure()

vsourcetoextPowerFailure()vsourcetoget_extPowerFailure()

Returns true if external power supply voltage is too low.

js	function get_extPowerFailure()
php	function get_extPowerFailure()
cpp	Y_EXTPOWERFAILURE_enum get_extPowerFailure()
m	-(Y_EXTPOWERFAILURE_enum) extPowerFailure
pas	function get_extPowerFailure(): Integer
vb	function get_extPowerFailure() As Integer
cs	int get_extPowerFailure()
java	int get_extPowerFailure()
py	def get_extPowerFailure()
cmd	YVSource target get_extPowerFailure

Returns :

either Y_EXTPOWERFAILURE_FALSE or Y_EXTPOWERFAILURE_TRUE, according to true if external power supply voltage is too low

On failure, throws an exception or returns Y_EXTPOWERFAILURE_INVALID.

vsource→get_failure()**YVSource****vsource→failure()****vsource→get_failure()**

Returns true if the module is in failure mode.

`Y_FAILURE_enum get_failure()`

vsource→get_failure()**vsource→failure()****vsource→get_failure()**

Returns true if the module is in failure mode.

<code>js</code>	<code>function get_failure()</code>
<code>php</code>	<code>function get_failure()</code>
<code>cpp</code>	<code>Y_FAILURE_enum get_failure()</code>
<code>m</code>	<code>-(Y_FAILURE_enum) failure</code>
<code>pas</code>	<code>function get_failure(): Integer</code>
<code>vb</code>	<code>function get_failure() As Integer</code>
<code>cs</code>	<code>int get_failure()</code>
<code>java</code>	<code>int get_failure()</code>
<code>py</code>	<code>def get_failure()</code>
<code>cmd</code>	<code>YVSource target get_failure</code>

More information can be obtained by testing `get_overheat`, `get_overcurrent` etc... When a error condition is met, the output voltage is set to zéro and cannot be changed until the `reset()` function is called.

Returns :

either `Y_FAILURE_FALSE` or `Y_FAILURE_TRUE`, according to true if the module is in failure mode

On failure, throws an exception or returns `Y_FAILURE_INVALID`.

vsourc→**get_friendlyName()****YVSource****vsourc**→**friendlyName()****vsourc**→
get_friendlyName()Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.virtual string **get_friendlyName()****vsourc**→**get_friendlyName()****vsourc**→**friendlyName()****vsourc**→**get_friendlyName()**Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.`js` function **get_friendlyName()**`php` function **get_friendlyName()**`cpp` virtual string **get_friendlyName()**`m` `-(NSString*) friendlyName``cs` override string **get_friendlyName()**`java` String **get_friendlyName()**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the function using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

vsource→**get_functionDescriptor()****YVSource****vsource**→**functionDescriptor()****vsource**→
get_vsourceDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()****vsource**→**get_functionDescriptor()****vsource**→**functionDescriptor()****vsource**→**get_vsourceDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor()
php	function get_functionDescriptor()
cpp	YFUN_DESCR get_functionDescriptor()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor() : YFUN_DESCR
vb	function get_functionDescriptor() As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor()
java	String get_functionDescriptor()
py	def get_functionDescriptor()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

vsSource→**get_functionId()****YVSource****vsSource**→**functionId()****vsSource**→**get_vsSourceId()**

Returns the hardware identifier of the function, without reference to the module.

```
string get_functionId( )
```

vsSource→**get_functionId()****vsSource**→**functionId()****vsSource**→**get_vsSourceId()**

Returns the hardware identifier of the function, without reference to the module.

js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()

For example `relay1`

Returns :

a string that identifies the function (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

vsourcesrc→get_hardwareId()**YVSource****vsourcesrc→hardwareId()**
vsourcesrc→get_hardwareId()

Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

string **get_hardwareId()**

vsourcesrc→get_hardwareId()**vsourcesrc→hardwareId()**
vsourcesrc→get_hardwareId()

Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

js	function get_hardwareId()
php	function get_hardwareId()
cpp	string get_hardwareId()
m	-(NSString*) hardwareId
vb	function get_hardwareId() As String
cs	string get_hardwareId()
java	String get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the function (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

vsource→**get_logicalName()****YVSource****vsource**→**logicalName()****vsource**→**get_logicalName()**

Returns the logical name of the voltage source.

string **get_logicalName()**

vsource→**get_logicalName()****vsource**→**logicalName()****vsource**→**get_logicalName()**

Returns the logical name of the voltage source.

js	function get_logicalName()
php	function get_logicalName()
cpp	string get_logicalName()
m	-(NSString*) logicalName
pas	function get_logicalName() : string
vb	function get_logicalName() As String
cs	string get_logicalName()
java	String get_logicalName()
py	def get_logicalName()
cmd	YVSource target get_logicalName

Returns :

a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

vsource→**get_module()****YVSource****vsource**→**module()****vsource**→**get_module()**

Gets the YModule object for the device on which the function is located.

```
YModule * get_module( )
```

vsource→**get_module()****vsource**→**module()****vsource**→**get_module()**

Gets the YModule object for the device on which the function is located.

js	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

vsSource→**get_overCurrent()****YVSource****vsSource**→**overCurrent()****vsSource**→
get_overCurrent()

Returns true if the appliance connected to the device is too greedy .

Y_OVERCURRENT_enum **get_overCurrent()****vsSource**→**get_overCurrent()****vsSource**→**overCurrent()****vsSource**→**get_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

js	function get_overCurrent()
php	function get_overCurrent()
cpp	Y_OVERCURRENT_enum get_overCurrent()
m	-(Y_OVERCURRENT_enum) overCurrent
pas	function get_overCurrent() : Integer
vb	function get_overCurrent() As Integer
cs	int get_overCurrent()
java	int get_overCurrent()
py	def get_overCurrent()
cmd	YVSource target get_overCurrent

Returns :

either **Y_OVERCURRENT_FALSE** or **Y_OVERCURRENT_TRUE**, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns **Y_OVERCURRENT_INVALID**.

vsource→**get_overHeat()****YVSource****vsource**→**overHeat()****vsource**→**get_overHeat()**

Returns TRUE if the module is overheating.

Y_OVERHEAT_enum **get_overHeat()****vsource**→**get_overHeat()****vsource**→**overHeat()****vsource**→**get_overHeat()**

Returns TRUE if the module is overheating.

js	function get_overHeat()
php	function get_overHeat()
cpp	Y_OVERHEAT_enum get_overHeat()
m	-(Y_OVERHEAT_enum) overHeat
pas	function get_overHeat() : Integer
vb	function get_overHeat() As Integer
cs	int get_overHeat()
java	int get_overHeat()
py	def get_overHeat()
cmd	YVSource target get_overHeat

Returns :

either Y_OVERHEAT_FALSE or Y_OVERHEAT_TRUE, according to TRUE if the module is overheating

On failure, throws an exception or returns Y_OVERHEAT_INVALID.

vsourc→**get_overLoad()****YVSource****vsourc**→**overLoad()****vsourc**→**get_overLoad()**

Returns true if the device is not able to maintaint the requested voltage output .

`Y_OVERLOAD_enum` **get_overLoad()**

vsourc→**get_overLoad()****vsourc**→**overLoad()****vsourc**→**get_overLoad()**

Returns true if the device is not able to maintaint the requested voltage output .

<code>js</code>	<code>function</code> get_overLoad()
<code>php</code>	<code>function</code> get_overLoad()
<code>cpp</code>	<code>Y_OVERLOAD_enum</code> get_overLoad()
<code>m</code>	<code>-(Y_OVERLOAD_enum)</code> <code>overLoad</code>
<code>pas</code>	<code>function</code> get_overLoad() : Integer
<code>vb</code>	<code>function</code> get_overLoad() As Integer
<code>cs</code>	<code>int</code> get_overLoad()
<code>java</code>	<code>int</code> get_overLoad()
<code>py</code>	<code>def</code> get_overLoad()
<code>cmd</code>	<code>YVSource</code> target get_overLoad

Returns :

either `Y_OVERLOAD_FALSE` or `Y_OVERLOAD_TRUE`, according to true if the device is not able to maintaint the requested voltage output

On failure, throws an exception or returns `Y_OVERLOAD_INVALID`.

vssource→get_regulationFailure()**YVSource****vs**source→regulationFailure()
vssource→get_regulationFailure()

Returns true if the voltage output is too high regarding the requested voltage .

Y_REGULATIONFAILURE_enum get_regulationFailure()

vssource→get_regulationFailure()**vs**source→regulationFailure()
vssource→get_regulationFailure()

Returns true if the voltage output is too high regarding the requested voltage .

js	function get_regulationFailure()
php	function get_regulationFailure()
cpp	Y_REGULATIONFAILURE_enum get_regulationFailure()
m	-(Y_REGULATIONFAILURE_enum) regulationFailure
pas	function get_regulationFailure(): Integer
vb	function get_regulationFailure() As Integer
cs	int get_regulationFailure()
java	int get_regulationFailure()
py	def get_regulationFailure()
cmd	YVSource target get_regulationFailure

Returns :

either Y_REGULATIONFAILURE_FALSE or Y_REGULATIONFAILURE_TRUE, according to true if the voltage output is too high regarding the requested voltage

On failure, throws an exception or returns Y_REGULATIONFAILURE_INVALID.

vsource→**get_unit()**
vsource→**unit()****vsource**→**get_unit()**

YVSource

Returns the measuring unit for the voltage.

string **get_unit()**

vsource→**get_unit()**
vsource→**unit()****vsource**→**get_unit()**

Returns the measuring unit for the voltage.

js	function get_unit()
php	function get_unit()
cpp	string get_unit()
m	-(NSString*) unit
pas	function get_unit() : string
vb	function get_unit() As String
cs	string get_unit()
java	String get_unit()
py	def get_unit()
cmd	YVSource target get_unit

Returns :
a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns Y_UNIT_INVALID.

vsource→**get_userdata()****YVSource****vsource**→**userData()****vsource**→**get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

vsource→**get_userdata()****vsource**→**userData()****vsource**→**get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

<code>js</code>	<code>function get_userdata()</code>
<code>php</code>	<code>function get_userdata()</code>
<code>cpp</code>	<code>void * get_userdata()</code>
<code>m</code>	<code>-(void*) userData</code>
<code>pas</code>	<code>function get_userdata(): Tobject</code>
<code>vb</code>	<code>function get_userdata() As Object</code>
<code>cs</code>	<code>object get_userdata()</code>
<code>java</code>	<code>Object get_userdata()</code>
<code>py</code>	<code>def get_userdata()</code>

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.

vssource→**get_voltage()**

YVSource

vssource→**voltage()**

vssource→**get_voltage()**

Returns the voltage output command (mV)

int **get_voltage()**

vssource→**get_voltage()**

vssource→**voltage()**

vssource→**get_voltage()**

Returns the voltage output command (mV)

js	function get_voltage()
php	function get_voltage()
cpp	int get_voltage()
m	-(int) voltage
pas	function get_voltage() : LongInt
vb	function get_voltage() As Integer
cs	int get_voltage()
java	int get_voltage()
py	def get_voltage()

Returns :
an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns Y_VOLTAGE_INVALID.

vsource→**isOnline()****vsource**→**isOnline()****YVSource**

Checks if the function is currently reachable, without raising any error.

```
bool isOnline( )
```

vsource→**isOnline()****vsource**→**isOnline()**

Checks if the function is currently reachable, without raising any error.

js	function isOnline ()
php	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	function isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	def isOnline ()

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :

`true` if the function can be reached, and `false` otherwise

vsourceto load()vsourceto load()**YVSource**

Preloads the function cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

vsourceto load()vsourceto load()

Preloads the function cache with a specified validity duration.

js	function load(msValidity)
php	function load(\$msValidity)
cpp	YRETCODE load(int msValidity)
m	-(YRETCODE) load : (int) msValidity
pas	function load(msValidity: integer): YRETCODE
vb	function load(ByVal msValidity As Integer) As YRETCODE
cs	YRETCODE load(int msValidity)
java	int load(long msValidity)
py	def load(msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

vsources→nextVSource()**vsources→nextVSource()**

YVSource

Continues the enumeration of voltage sources started using `yFirstVSource()`.

`YVSource * nextVSource()`

vsources→nextVSource()**vsources→nextVSource()**

Continues the enumeration of voltage sources started using `yFirstVSource()`.

<code>js</code>	<code>function nextVSource()</code>
<code>php</code>	<code>function nextVSource()</code>
<code>cpp</code>	<code>YVSource * nextVSource()</code>
<code>m</code>	<code>-(YVSource*) nextVSource</code>
<code>pas</code>	<code>function nextVSource(): TYVSource</code>
<code>vb</code>	<code>function nextVSource() As YVSource</code>
<code>cs</code>	<code>YVSource nextVSource()</code>
<code>java</code>	<code>YVSource nextVSource()</code>
<code>py</code>	<code>def nextVSource()</code>

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.

vsourc→pulse()**vsourc→pulse()****YVSource**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

```
int pulse( int voltage, int ms_duration)
```

vsourc→pulse()**vsourc→pulse()**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

js	function pulse(voltage, ms_duration)
php	function pulse(\$voltage, \$ms_duration)
cpp	int pulse(int voltage, int ms_duration)
m	-(int) pulse : (int) voltage : (int) ms_duration
pas	function pulse(voltage: integer, ms_duration: integer): integer
vb	function pulse(ByVal voltage As Integer, ByVal ms_duration As Integer) As Integer
cs	int pulse(int voltage, int ms_duration)
java	int pulse(int voltage, int ms_duration)
py	def pulse(voltage, ms_duration)
cmd	YVSource target pulse voltage ms_duration

Parameters :

voltage pulse voltage, in millivolts
ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsouce→registerValueCallback()**vsouce→**
registerValueCallback()

YVSource

Registers the callback function that is invoked on every change of advertised value.

```
void registerValueCallback( YDisplayUpdateCallback callback)
```

vsouce→registerValueCallback()**vsouce→registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

js	function registerValueCallback (callback)
php	function registerValueCallback (\$callback)
cpp	void registerValueCallback (YDisplayUpdateCallback callback)
pas	procedure registerValueCallback (callback : TGenericUpdateCallback)
vb	procedure registerValueCallback (ByVal callback As GenericUpdateCallback)
cs	void registerValueCallback (UpdateCallback callback)
java	void registerValueCallback (UpdateCallback callback)
py	def registerValueCallback (callback)
m	-(void) registerValueCallback : (YFunctionUpdateCallback) callback

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

vsSource→**set_logicalName()****YVSource****vsSource**→**setLogicalName()****vsSource**→**set_logicalName()**

Changes the logical name of the voltage source.

int **set_logicalName**(const string& **newval**)**vsSource**→**set_logicalName()****vsSource**→**setLogicalName()****vsSource**→**set_logicalName()**

Changes the logical name of the voltage source.

js	function set_logicalName (newval)
php	function set_logicalName (\$newval)
cpp	int set_logicalName (const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
py	def set_logicalName (newval)
cmd	YVSource target set_logicalName newval

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :**newval** a string corresponding to the logical name of the voltage source**Returns :**

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→set_userdata()**YVSource****vsource→setUserData()****vsource→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

vsource→set_userdata()**vsource→setUserData()****vsource→set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

js	function set_userdata (data)
php	function set_userdata (\$data)
cpp	void set_userdata (void* data)
m	-(void) setUserData : (void*) data
pas	procedure set_userdata (data : Tobject)
vb	procedure set_userdata (ByVal data As Object)
cs	void set_userdata (object data)
java	void set_userdata (Object data)
py	def set_userdata (data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

vsourceset_voltage()**YVSource****vsourcesetVoltage()vsourceset_voltage()**

Tunes the device output voltage (milliVolts).

```
int set_voltage( int newval)
```

vsourceset_voltage()**vsourcesetVoltage()vsourceset_voltage()**

Tunes the device output voltage (milliVolts).

js	function set_voltage(newval)
php	function set_voltage(\$newval)
cpp	int set_voltage(int newval)
m	-(int) setVoltage : (int) newval
pas	function set_voltage(newval: LongInt): integer
vb	function set_voltage(ByVal newval As Integer) As Integer
cs	int set_voltage(int newval)
java	int set_voltage(int newval)
py	def set_voltage(newval)
cmd	YVSource target set_voltage newval

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→voltageMove()**vsource→voltageMove ()****YVSource**

Performs a smooth move at constant speed toward a given value.

```
int voltageMove( int target, int ms_duration)
```

vsource→voltageMove()**vsource→voltageMove ()**

Performs a smooth move at constant speed toward a given value.

js	function voltageMove (target , ms_duration)
php	function voltageMove (\$target , \$ms_duration)
cpp	int voltageMove (int target , int ms_duration)
m	-(int) voltageMove : (int) target : (int) ms_duration
pas	function voltageMove (target : integer, ms_duration : integer): integer
vb	function voltageMove (ByVal target As Integer, ByVal ms_duration As Integer) As Integer
cs	int voltageMove (int target , int ms_duration)
java	int voltageMove (int target , int ms_duration)
py	def voltageMove (target , ms_duration)
cmd	YVSource target voltageMove target ms_duration

Parameters :

target new output value at end of transition, in milliVolts.
ms_duration transition duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.43. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
php	require_once('yocto_wakeupmonitor.php');
c++	#include "yocto_wakeupmonitor.h"
m	#import "yocto_wakeupmonitor.h"
pas	uses yocto_wakeupmonitor;
vb	yocto_wakeupmonitor.vb
cs	yocto_wakeupmonitor.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py	from yocto_wakeupmonitor import *

Global functions
yFindWakeUpMonitor(func) Retrieves a monitor for a given identifier.
yFirstWakeUpMonitor() Starts the enumeration of monitors currently accessible.
YWakeupMonitor methods
wakeupmonitor→describe() Returns a short text that describes unambiguously the instance of the monitor in the form TYPE (NAME) =SERIAL . FUNCTIONID.
wakeupmonitor→get_advertisedValue() Returns the current value of the monitor (no more than 6 characters).
wakeupmonitor→get_errorMessage() Returns the error message of the latest error with the monitor.
wakeupmonitor→get_errorType() Returns the numerical error code of the latest error with the monitor.
wakeupmonitor→get_friendlyName() Returns a global identifier of the monitor in the format MODULE_NAME . FUNCTION_NAME.
wakeupmonitor→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
wakeupmonitor→get_functionId() Returns the hardware identifier of the monitor, without reference to the module.
wakeupmonitor→get_hardwareId() Returns the unique hardware identifier of the monitor in the form SERIAL . FUNCTIONID.
wakeupmonitor→get_logicalName() Returns the logical name of the monitor.
wakeupmonitor→get_module() Gets the YModule object for the device on which the function is located.
wakeupmonitor→get_module_async(callback, context) Gets the YModule object for the device on which the function is located (asynchronous version).
wakeupmonitor→get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)
wakeupmonitor→get_powerDuration() Returns the maximal wake up time (in seconds) before automatically going to sleep.
wakeupmonitor→get_sleepCountdown() Returns the delay before the next sleep period.
wakeupmonitor→get_userData() Returns the value of the userData attribute, as previously stored using method set_userData.
wakeupmonitor→get_wakeUpReason() Returns the latest wake up reason.
wakeupmonitor→get_wakeUpState() Returns the current state of the monitor
wakeupmonitor→isOnline() Checks if the monitor is currently reachable, without raising any error.
wakeupmonitor→isOnline_async(callback, context) Checks if the monitor is currently reachable, without raising any error (asynchronous version).
wakeupmonitor→load(msValidity) Preloads the monitor cache with a specified validity duration.
wakeupmonitor→load_async(msValidity, callback, context) Preloads the monitor cache with a specified validity duration (asynchronous version).
wakeupmonitor→nextWakeUpMonitor() Continues the enumeration of monitors started using yFirstWakeUpMonitor().
wakeupmonitor→registerValueCallback(callback) Registers the callback function that is invoked on every change of advertised value.
wakeupmonitor→resetSleepCountDown() Resets the sleep countdown.
wakeupmonitor→set_logicalName(newval) Changes the logical name of the monitor.
wakeupmonitor→set_nextWakeUp(newval) Changes the days of the week when a wake up must take place.
wakeupmonitor→set_powerDuration(newval) Changes the maximal wake up time (seconds) before automatically going to sleep.
wakeupmonitor→set_sleepCountdown(newval) Changes the delay before the next sleep period.
wakeupmonitor→set_userData(data) Stores a user context provided as argument in the userData attribute of the function.
wakeupmonitor→sleep(secBeforeSleep) Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.
wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep) Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.
wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep) Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.
wakeupmonitor→wait_async(callback, context)

3. Reference

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wakeupmonitor→wakeUp()

Forces a wake up.

YWakeUpMonitor.FindWakeUpMonitor() yFindWakeUpMonitor()yFindWakeUpMonitor()

YWakeUpMonitor

Retrieves a monitor for a given identifier.

```
YWakeUpMonitor* yFindWakeUpMonitor( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpMonitor.isOnline()` to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the monitor

Returns :

a `YWakeUpMonitor` object allowing you to drive the monitor.

YWakeUpMonitor.FirstWakeUpMonitor()

YWakeUpMonitor

yFirstWakeUpMonitor()`yFirstWakeUpMonitor()`

Starts the enumeration of monitors currently accessible.

`YWakeUpMonitor* yFirstWakeUpMonitor()`

Use the method `YWakeUpMonitor.nextWakeUpMonitor()` to iterate on next monitors.

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to the first monitor currently online, or a `null` pointer if there are none.

wakeupmonitor→describe()wakeupmonitor→
describe()

YWakeUpMonitor

Returns a short text that describes unambiguously the instance of the monitor in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the monitor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupmonitor→**get_advertisedValue()**

YWakeUpMonitor

wakeupmonitor→**advertisedValue()****wakeupmonitor**

→**get_advertisedValue()**

Returns the current value of the monitor (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the monitor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wakeupmonitor→get_errorMessage()**YWakeUpMonitor****wakeupmonitor→errorMessage()****wakeupmonitor→**
get_errorMessage()

Returns the error message of the latest error with the monitor.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the monitor object

wakeupmonitor→get_errorType()

YWakeUpMonitor

wakeupmonitor→errorType()**wakeupmonitor→**

get_errorType()

Returns the numerical error code of the latest error with the monitor.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the monitor object

wakeupmonitor→**get_friendlyName()****YWakeUpMonitor****wakeupmonitor**→**friendlyName()****wakeupmonitor**→
get_friendlyName()

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the monitor if they are defined, otherwise the serial number of the module and the hardware identifier of the monitor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the monitor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wakeupmonitor→get_functionDescriptor()

YWakeUpMonitor

wakeupmonitor→functionDescriptor()

wakeupmonitor→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR [get_functionDescriptor\(\)](#) ()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupmonitor→**get_functionId()****YWakeUpMonitor****wakeupmonitor**→**functionId()****wakeupmonitor**→
get_functionId()

Returns the hardware identifier of the monitor, without reference to the module.

`string` **get_functionId()**

For example `relay1`

Returns :

a string that identifies the monitor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupmonitor→get_hardwareId()

YWakeUpMonitor

wakeupmonitor→hardwareId()
wakeupmonitor→
get_hardwareId()

Returns the unique hardware identifier of the monitor in the form SERIAL.FUNCTIONID.

string get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the monitor. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the monitor (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

wakeupmonitor→**get_logicalName()****YWakeUpMonitor****wakeupmonitor**→**logicalName()****wakeupmonitor**→
get_logicalName()

Returns the logical name of the monitor.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the monitor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wakeupmonitor→get_module()

YWakeUpMonitor

wakeupmonitor→module()wakeupmonitor→
get_module()

Gets the YModule object for the device on which the function is located.

YModule * get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

wakeupmonitor→**get_nextWakeUp()****YWakeUpMonitor****wakeupmonitor**→**nextWakeUp()****wakeupmonitor**→
get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)**s64** **get_nextWakeUp()****Returns :**

an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns `Y_NEXTWAKEUP_INVALID`.

wakeupmonitor→get_powerDuration()

YWakeUpMonitor

wakeupmonitor→powerDuration()
wakeupmonitor→
get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

`int get_powerDuration()`

Returns :

an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns Y_POWERDURATION_INVALID.

wakeupmonitor→**get_sleepCountdown()****YWakeUpMonitor****wakeupmonitor**→**sleepCountdown()****wakeupmonitor**→**get_sleepCountdown()**

Returns the delay before the next sleep period.

int **get_sleepCountdown()**

Returns :

an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns Y_SLEEP_COUNTDOWN_INVALID.

wakeupmonitor→get_userdata()

YWakeUpMonitor

wakeupmonitor→userdata()wakeupmonitor→
get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

wakeupmonitor→**get_wakeUpReason()****YWakeUpMonitor****wakeupmonitor**→**wakeUpReason()****wakeupmonitor**→
get_wakeUpReason()

Returns the latest wake up reason.

Y_WAKEUPREASON_enum **get_wakeUpReason()****Returns :**

a value among Y_WAKEUPREASON_USBPOWER, Y_WAKEUPREASON_EXTPOWER,
 Y_WAKEUPREASON_ENDOFSLEEP, Y_WAKEUPREASON_EXTSIG1,
 Y_WAKEUPREASON_EXTSIG2, Y_WAKEUPREASON_EXTSIG3,
 Y_WAKEUPREASON_EXTSIG4, Y_WAKEUPREASON_SCHEDULE1,
 Y_WAKEUPREASON_SCHEDULE2, Y_WAKEUPREASON_SCHEDULE3,
 Y_WAKEUPREASON_SCHEDULE4, Y_WAKEUPREASON_SCHEDULE5 and
 Y_WAKEUPREASON_SCHEDULE6 corresponding to the latest wake up reason

On failure, throws an exception or returns Y_WAKEUPREASON_INVALID.

wakeupmonitor→**get_wakeUpState()**

YWakeUpMonitor

wakeupmonitor→**wakeUpState()****wakeupmonitor**→
get_wakeUpState()

Returns the current state of the monitor

Y_WAKEUPSTATE_enum **get_wakeUpState()**

Returns :

either **Y_WAKEUPSTATE_SLEEPING** or **Y_WAKEUPSTATE_AWAKE**, according to the current state of the monitor

On failure, throws an exception or returns **Y_WAKEUPSTATE_INVALID**.

wakeupmonitor→**isOnline()****wakeupmonitor**→
isOnline()

YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error.

bool isOnline()

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

Returns :

`true` if the monitor can be reached, and `false` otherwise

wakeupmonitor→**load()****wakeupmonitor**→**load()****YWakeUpMonitor**

Preloads the monitor cache with a specified validity duration.

YRETCODE **load(int msValidity)**

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→nextWakeUpMonitor()**YWakeUpMonitor****wakeupmonitor→nextWakeUpMonitor()**

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

`YWakeUpMonitor * nextWakeUpMonitor()`

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a `null` pointer if there are no more monitors to enumerate.

wakeupmonitor→registerValueCallback()**YWakeUpMonitor****wakeupmonitor→registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YWakeUpMonitorValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupmonitor→resetSleepCountDown()**YWakeUpMonitor****wakeupmonitor→resetSleepCountDown()**

Resets the sleep countdown.

int resetSleepCountDown()

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→**set_logicalName()****YWakeUpMonitor****wakeupmonitor**→**setLogicalName()****wakeupmonitor**
→**set_logicalName()**

Changes the logical name of the monitor.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the monitor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→**set_nextWakeUp()****YWakeUpMonitor****wakeupmonitor**→**setNextWakeUp()****wakeupmonitor**→**set_nextWakeUp()**

Changes the days of the week when a wake up must take place.

```
int set_nextWakeUp( s64 newval)
```

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_powerDuration()

YWakeUpMonitor

wakeupmonitor→setPowerDuration()

wakeupmonitor→set_powerDuration()

Changes the maximal wake up time (seconds) before automatically going to sleep.

```
int set_powerDuration( int newval)
```

Parameters :

newval an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_sleepCountdown()**YWakeUpMonitor****wakeupmonitor→setSleepCountdown()****wakeupmonitor→set_sleepCountdown()**

Changes the delay before the next sleep period.

```
int set_sleepCountdown( int newval)
```

Parameters :

newval an integer corresponding to the delay before the next sleep period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_userdata()

YWakeUpMonitor

wakeupmonitor→setUserData()
wakeupmonitor→set_userdata()

Stores a user context provided as argument in the `userData` attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

wakeupmonitor→sleep()**wakeupmonitor→sleep()****YWakeUpMonitor**

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

```
int sleep( int secBeforeSleep)
```

Parameters :

secBeforeSleep number of seconds before going into sleep mode,

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepFor()
wakeupmonitor→sleepFor()

YWakeUpMonitor

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
int sleepFor( int secUntilWakeUp, int secBeforeSleep)
```

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

secUntilWakeUp sleep duration, in secondes

secBeforeSleep number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→**sleepUntil()****wakeupmonitor**→
sleepUntil()

YWakeUpMonitor

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
int sleepUntil( int wakeUpTime, int secBeforeSleep)
```

The count down before sleep can be canceled with `resetSleepCountDown`.

Parameters :

wakeUpTime wake-up datetime (UNIX format)

secBeforeSleep number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→wakeUp()wakeupmonitor→
wakeUp()

YWakeUpMonitor

Forces a wake up.

```
int wakeUp( )
```

3.44. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupschedule.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWakeUpSchedule = yoctolib.YWakeUpSchedule;
php	require_once('yocto_wakeupschedule.php');
c++	#include "yocto_wakeupschedule.h"
m	#import "yocto_wakeupschedule.h"
pas	uses yocto_wakeupschedule;
vb	yocto_wakeupschedule.vb
cs	yocto_wakeupschedule.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
py	from yocto_wakeupschedule import *

Global functions

yFindWakeUpSchedule(func)

Retrieves a wake up schedule for a given identifier.

yFirstWakeUpSchedule()

Starts the enumeration of wake up schedules currently accessible.

YWakeUpSchedule methods

wakeupschedule→describe()

Returns a short text that describes unambiguously the instance of the wake up schedule in the form TYPE (NAME) = SERIAL . FUNCTIONID.

wakeupschedule→get_advertisedValue()

Returns the current value of the wake up schedule (no more than 6 characters).

wakeupschedule→get_errorMessage()

Returns the error message of the latest error with the wake up schedule.

wakeupschedule→get_errorType()

Returns the numerical error code of the latest error with the wake up schedule.

wakeupschedule→get_friendlyName()

Returns a global identifier of the wake up schedule in the format MODULE_NAME . FUNCTION_NAME.

wakeupschedule→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

wakeupschedule→get_functionId()

Returns the hardware identifier of the wake up schedule, without reference to the module.

wakeupschedule→get_hardwareId()

Returns the unique hardware identifier of the wake up schedule in the form SERIAL . FUNCTIONID.

wakeupschedule→get_hours()

Returns the hours scheduled for wake up.

wakeupschedule→get_logicalName()

Returns the logical name of the wake up schedule.

wakeupschedule→get_minutes()

Returns all the minutes of each hour that are scheduled for wake up.

wakeupschedule→get_minutesA()

	Returns the minutes in the 00-29 interval of each hour scheduled for wake up.
wakeupschedule→get_minutesB()	Returns the minutes in the 30-59 interval of each hour scheduled for wake up.
wakeupschedule→get_module()	Gets the YModule object for the device on which the function is located.
wakeupschedule→get_module_async(callback, context)	Gets the YModule object for the device on which the function is located (asynchronous version).
wakeupschedule→get_monthDays()	Returns the days of the month scheduled for wake up.
wakeupschedule→get_months()	Returns the months scheduled for wake up.
wakeupschedule→get_nextOccurence()	Returns the date/time (seconds) of the next wake up occurrence
wakeupschedule→get_userData()	Returns the value of the userData attribute, as previously stored using method set_userData.
wakeupschedule→get_weekDays()	Returns the days of the week scheduled for wake up.
wakeupschedule→isOnline()	Checks if the wake up schedule is currently reachable, without raising any error.
wakeupschedule→isOnline_async(callback, context)	Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).
wakeupschedule→load(msValidity)	Preloads the wake up schedule cache with a specified validity duration.
wakeupschedule→load_async(msValidity, callback, context)	Preloads the wake up schedule cache with a specified validity duration (asynchronous version).
wakeupschedule→nextWakeUpSchedule()	Continues the enumeration of wake up schedules started using yFirstWakeUpSchedule().
wakeupschedule→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
wakeupschedule→set_hours(newval)	Changes the hours when a wake up must take place.
wakeupschedule→set_logicalName(newval)	Changes the logical name of the wake up schedule.
wakeupschedule→set_minutes(bitmap)	Changes all the minutes where a wake up must take place.
wakeupschedule→set_minutesA(newval)	Changes the minutes in the 00-29 interval when a wake up must take place.
wakeupschedule→set_minutesB(newval)	Changes the minutes in the 30-59 interval when a wake up must take place.
wakeupschedule→set_monthDays(newval)	Changes the days of the month when a wake up must take place.
wakeupschedule→set_months(newval)	Changes the months when a wake up must take place.
wakeupschedule→set_userData(data)	Stores a user context provided as argument in the userData attribute of the function.

wakeupschedule→**set_weekDays**(**newval**)

Changes the days of the week when a wake up must take place.

wakeupschedule→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWakeUpSchedule.FindWakeUpSchedule() yFindWakeUpSchedule()yFindWakeUpSchedule()

YWakeUpSchedule

Retrieves a wake up schedule for a given identifier.

```
YWakeUpSchedule* yFindWakeUpSchedule( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpSchedule.isOnline()` to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the wake up schedule

Returns :

a `YWakeUpSchedule` object allowing you to drive the wake up schedule.

YWakeUpSchedule.FirstWakeUpSchedule()
yFirstWakeUpSchedule()**yFirstWakeUpSchedule()****YWakeUpSchedule**

Starts the enumeration of wake up schedules currently accessible.

YWakeUpSchedule* yFirstWakeUpSchedule()

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to the first wake up schedule currently online,
or a `null` pointer if there are none.

wakeupschedule→**describe()****wakeupschedule**→
describe()

YWakeUpSchedule

Returns a short text that describes unambiguously the instance of the wake up schedule in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the wake up schedule (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupschedule→get_advertisedValue()**YWakeUpSchedule****wakeupschedule→advertisedValue()****wakeupschedule→get_advertisedValue()**

Returns the current value of the wake up schedule (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the wake up schedule (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wakeupschedule→get_errorMessage()

YWakeUpSchedule

wakeupschedule→errorMessage()wakeupschedule

→get_errorMessage()

Returns the error message of the latest error with the wake up schedule.

string get_errorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the wake up schedule object

wakeupschedule→get_errorType()**YWakeUpSchedule****wakeupschedule→errorType()**
wakeupschedule→get_errorType()

Returns the numerical error code of the latest error with the wake up schedule.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the wake up schedule object

wakeupschedule→get_friendlyName()

YWakeUpSchedule

wakeupschedule→friendlyName()
wakeupschedule→get_friendlyName()

Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the wake up schedule if they are defined, otherwise the serial number of the module and the hardware identifier of the wake up schedule (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the wake up schedule using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wakeupschedule→get_functionDescriptor()**YWakeUpSchedule****wakeupschedule→functionDescriptor()****wakeupschedule→get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupschedule→**get_functionId()**

YWakeUpSchedule

wakeupschedule→**functionId()****wakeupschedule**→
get_functionId()

Returns the hardware identifier of the wake up schedule, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the wake up schedule (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupschedule→get_hardwareId()**YWakeUpSchedule****wakeupschedule→hardwareId()**
wakeupschedule→get_hardwareId()

Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.

```
string get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wake up schedule. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the wake up schedule (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

wakeupschedule→get_hours()

YWakeUpSchedule

wakeupschedule→hours()**wakeupschedule→**
get_hours ()

Returns the hours scheduled for wake up.

```
int get_hours( )
```

Returns :

an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns Y_HOURS_INVALID.

wakeupschedule→get_logicalName()**YWakeUpSchedule****wakeupschedule→logicalName()**
wakeupschedule→get_logicalName()

Returns the logical name of the wake up schedule.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the wake up schedule. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wakeupschedule→get_minutes()

YWakeUpSchedule

wakeupschedule→minutes()**wakeupschedule→**
get_minutes()

Returns all the minutes of each hour that are scheduled for wake up.

s64 **get_minutes()**

wakeupschedule→get_minutesA()**YWakeUpSchedule****wakeupschedule→minutesA()****wakeupschedule→**
get_minutesA()

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

int **get_minutesA()** ()

Returns :

an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y_MINUTESA_INVALID.

wakeupschedule→get_minutesB()

YWakeUpSchedule

wakeupschedule→minutesB()**wakeupschedule→**
get_minutesB()

Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

```
int get_minutesB( )
```

Returns :

an integer corresponding to the minutes in the 30-59 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y_MINUTESB_INVALID.

wakeupschedule→get_module()**YWakeUpSchedule****wakeupschedule→module()****wakeupschedule→**
get_module()

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

wakeupschedule→**get_monthDays()**

YWakeUpSchedule

wakeupschedule→**monthDays()****wakeupschedule**→
get_monthDays ()

Returns the days of the month scheduled for wake up.

int **get_monthDays()**

Returns :

an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns Y_MONTHDAYS_INVALID.

wakeupschedule→get_months()**YWakeUpSchedule****wakeupschedule→months()****wakeupschedule→**
get_months()

Returns the months scheduled for wake up.

int **get_months()**

Returns :

an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns Y_MONTHS_INVALID.

wakeupschedule→**get_nextOccurence()**

YWakeUpSchedule

wakeupschedule→**nextOccurence()****wakeupschedule**

→**get_nextOccurence()**

Returns the date/time (seconds) of the next wake up occurrence

s64 **get_nextOccurence()**

Returns :

an integer corresponding to the date/time (seconds) of the next wake up occurrence

On failure, throws an exception or returns Y_NEXTOCCURENCE_INVALID.

wakeupschedule→get_userdata()**YWakeUpSchedule****wakeupschedule→userdata()****wakeupschedule→**
get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

wakeupschedule→get_weekDays()

YWakeUpSchedule

wakeupschedule→weekDays()**wakeupschedule→**
get_weekDays ()

Returns the days of the week scheduled for wake up.

int get_weekDays()

Returns :

an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns Y_WEEKDAYS_INVALID.

wakeupschedule→**isOnline()****wakeupschedule**→
isOnline()

YWakeUpSchedule

Checks if the wake up schedule is currently reachable, without raising any error.

bool **isOnline()**

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

Returns :

`true` if the wake up schedule can be reached, and `false` otherwise

wakeupschedule→load()**wakeupschedule→load()****YWakeUpSchedule**

Preloads the wake up schedule cache with a specified validity duration.

YRETCODE **load**(int **msValidity**)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→nextWakeUpSchedule()**YWakeUpSchedule****wakeupschedule→nextWakeUpSchedule()**

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

`YWakeupSchedule * nextWakeUpSchedule()`

Returns :

a pointer to a `YWakeupSchedule` object, corresponding to a wake up schedule currently online, or a `null` pointer if there are no more wake up schedules to enumerate.

wakeupschedule→registerValueCallback()

YWakeUpSchedule

wakeupschedule→registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YWakeUpScheduleValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupschedule→set_hours()**YWakeUpSchedule****wakeupschedule→setHours()****wakeupschedule→**
set_hours()

Changes the hours when a wake up must take place.

```
int set_hours( int newval)
```

Parameters :

newval an integer corresponding to the hours when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_logicalName()

YWakeUpSchedule

wakeupschedule→setLogicalName()

wakeupschedule→set_logicalName()

Changes the logical name of the wake up schedule.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wake up schedule.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutes()**YWakeUpSchedule****wakeupschedule→setMinutes()**
wakeupschedule→set_minutes()

Changes all the minutes where a wake up must take place.

```
int set_minutes( s64 bitmap)
```

Parameters :

bitmap Minutes 00-59 of each hour scheduled for wake up.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→**set_minutesA()**

YWakeUpSchedule

wakeupschedule→**setMinutesA()****wakeupschedule**→
set_minutesA()

Changes the minutes in the 00-29 interval when a wake up must take place.

```
int set_minutesA( int newval)
```

Parameters :

newval an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesB()**YWakeUpSchedule****wakeupschedule→setMinutesB()****wakeupschedule→**
set_minutesB()

Changes the minutes in the 30-59 interval when a wake up must take place.

```
int set_minutesB( int newval)
```

Parameters :

newval an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→**set_monthDays()**

YWakeUpSchedule

wakeupschedule→**setMonthDays()****wakeupschedule**

→**set_monthDays ()**

Changes the days of the month when a wake up must take place.

```
int set_monthDays( int newval)
```

Parameters :

newval an integer corresponding to the days of the month when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→**set_months()****YWakeUpSchedule****wakeupschedule**→**setMonths()****wakeupschedule**→
set_months()

Changes the months when a wake up must take place.

```
int set_months( int newval)
```

Parameters :

newval an integer corresponding to the months when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_userdata()

YWakeUpSchedule

wakeupschedule→setUserData()**wakeupschedule→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

wakeupschedule→**set_weekDays()****YWakeUpSchedule****wakeupschedule**→**setWeekDays()****wakeupschedule**→**set_weekDays ()**

Changes the days of the week when a wake up must take place.

```
int set_weekDays( int newval)
```

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.45. Watchdog function interface

The watchdog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven directly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_watchdog.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWatchdog = yoctolib.YWatchdog;
php	require_once('yocto_watchdog.php');
c++	#include "yocto_watchdog.h"
m	#import "yocto_watchdog.h"
pas	uses yocto_watchdog;
vb	yocto_watchdog.vb
cs	yocto_watchdog.cs
java	import com.yoctopuce.YoctoAPI.YWatchdog;
py	from yocto_watchdog import *

Global functions

yFindWatchdog(func)

Retrieves a watchdog for a given identifier.

yFirstWatchdog()

Starts the enumeration of watchdog currently accessible.

YWatchdog methods

watchdog→delayedPulse(ms_delay, ms_duration)

Schedules a pulse.

watchdog→describe()

Returns a short text that describes unambiguously the instance of the watchdog in the form
TYPE (NAME) = SERIAL . FUNCTIONID.

watchdog→get_advertisedValue()

Returns the current value of the watchdog (no more than 6 characters).

watchdog→get_autoStart()

Returns the watchdog running state at module power on.

watchdog→get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

watchdog→get_errorMessage()

Returns the error message of the latest error with the watchdog.

watchdog→get_errorType()

Returns the numerical error code of the latest error with the watchdog.

watchdog→get_friendlyName()

Returns a global identifier of the watchdog in the format MODULE_NAME . FUNCTION_NAME.

watchdog→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

watchdog→get_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

watchdog→get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form SERIAL . FUNCTIONID.

watchdog→get_logicalName()

Returns the logical name of the watchdog.

watchdog→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

watchdog→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

watchdog→get_module()

Gets the YModule object for the device on which the function is located.

watchdog→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

watchdog→get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

watchdog→get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

watchdog→get_running()

Returns the watchdog running state.

watchdog→get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

watchdog→get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

watchdog→get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

watchdog→get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

watchdog→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

watchdog→isOnline()

Checks if the watchdog is currently reachable, without raising any error.

watchdog→isOnline_async(callback, context)

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

watchdog→load(msValidity)

Preloads the watchdog cache with a specified validity duration.

watchdog→load_async(msValidity, callback, context)

Preloads the watchdog cache with a specified validity duration (asynchronous version).

watchdog→nextWatchdog()

Continues the enumeration of watchdog started using yFirstWatchdog().

watchdog→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

watchdog→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

watchdog→resetWatchdog()

Resets the watchdog.

watchdog→set_autoStart(newval)

Changes the watchdog running state at module power on.

watchdog→set_logicalName(newval)

Changes the logical name of the watchdog.

watchdog→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

watchdog→set_maxTimeOnStateB(newval)

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

watchdog→set_output(newval)

Changes the output state of the watchdog, when used as a simple switch (single throw).

watchdog→set_running(newval)

Changes the running state of the watchdog.

watchdog→set_state(newval)

Changes the state of the watchdog (A for the idle position, B for the active position).

watchdog→set_stateAtPowerOn(newval)

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

watchdog→set_triggerDelay(newval)

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

watchdog→set_triggerDuration(newval)

Changes the duration of resets caused by the watchdog, in milliseconds.

watchdog→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

watchdog→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWatchdog.FindWatchdog() yFindWatchdog()yFindWatchdog()

YWatchdog

Retrieves a watchdog for a given identifier.

```
YWatchdog* yFindWatchdog( const string& func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWatchdog.isOnline()` to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the watchdog

Returns :

a `YWatchdog` object allowing you to drive the watchdog.

YWatchdog.FirstWatchdog()

YWatchdog

yFirstWatchdog()`yFirstWatchdog()`

Starts the enumeration of watchdog currently accessible.

`YWatchdog* yFirstWatchdog()`

Use the method `YWatchdog.nextWatchdog()` to iterate on next watchdog.

Returns :

a pointer to a `YWatchdog` object, corresponding to the first watchdog currently online, or a `null` pointer if there are none.

watchdog→delayedPulse() watchdog→
delayedPulse()

YWatchdog

Schedules a pulse.

```
int delayedPulse( int ms_delay, int ms_duration)
```

Parameters :

ms_delay waiting time before the pulse, in milliseconds

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→describe()**YWatchdog**

Returns a short text that describes unambiguously the instance of the watchdog in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the watchdog (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

watchdog→get_advertisedValue()**YWatchdog****watchdog→advertisedValue()****watchdog→****get_advertisedValue()**

Returns the current value of the watchdog (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the watchdog (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

watchdog→**get_autoStart()**

YWatchdog

watchdog→**autoStart()****watchdog**→

get_autoStart()

Returns the watchdog running state at module power on.

[Y_AUTOSTART_enum](#) **get_autoStart()**

Returns :

either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog running state at module power on

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

watchdog→get_countdown()**YWatchdog****watchdog→countdown()watchdog→****get_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

s64 **get_countdown()**

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns Y_COUNTDOWN_INVALID.

watchdog→**get_errorMessage()**

YWatchdog

watchdog→**errorMessage()****watchdog**→

get_errorMessage()

Returns the error message of the latest error with the watchdog.

string **get_errorMessage()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the watchdog object

watchdog→get_errorType()**YWatchdog****watchdog→errorType()**
watchdog→get_errorType()

Returns the numerical error code of the latest error with the watchdog.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the watchdog object

watchdog→**get_friendlyName()**

YWatchdog

watchdog→**friendlyName()****watchdog**→

get_friendlyName()

Returns a global identifier of the watchdog in the format `MODULE_NAME . FUNCTION_NAME`.

`string get_friendlyName()`

The returned string uses the logical names of the module and of the watchdog if they are defined, otherwise the serial number of the module and the hardware identifier of the watchdog (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the watchdog using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

watchdog→get_functionDescriptor()**YWatchdog****watchdog→functionDescriptor()****watchdog→****get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

watchdog→**get_functionId()**

YWatchdog

watchdog→**functionId()****watchdog**→

get_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the watchdog (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

watchdog→get_hardwareId()**YWatchdog****watchdog→hardwareId()watchdog→****get_hardwareId()**

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

```
string get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the watchdog (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

watchdog→**get_logicalName()**

YWatchdog

watchdog→**logicalName()****watchdog**→
get_logicalName()

Returns the logical name of the watchdog.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the watchdog. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

watchdog→get_maxTimeOnStateA()**YWatchdog****watchdog→maxTimeOnStateA()****watchdog→****get_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

s64 **get_maxTimeOnStateA()**

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.

watchdog→get_maxTimeOnStateB()

YWatchdog

watchdog→maxTimeOnStateB() watchdog→

get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

s64 **get_maxTimeOnStateB()**

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEB_INVALID.

watchdog→get_module()**YWatchdog****watchdog→module()****watchdog→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

watchdog→get_output()

YWatchdog

watchdog→output()**watchdog→get_output()**

Returns the output state of the watchdog, when used as a simple switch (single throw).

`Y_OUTPUT_enum` **get_output()**

Returns :

either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns `Y_OUTPUT_INVALID`.

watchdog→get_pulseTimer()**YWatchdog****watchdog→pulseTimer()watchdog→
get_pulseTimer()**

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

s64 **get_pulseTimer()**

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

watchdog→get_running()

YWatchdog

watchdog→running()**watchdog→get_running()**

Returns the watchdog running state.

`Y_RUNNING_enum` **get_running()**

Returns :

either `Y_RUNNING_OFF` or `Y_RUNNING_ON`, according to the watchdog running state

On failure, throws an exception or returns `Y_RUNNING_INVALID`.

watchdog→get_state()**YWatchdog****watchdog→state()****watchdog→get_state()**

Returns the state of the watchdog (A for the idle position, B for the active position).

`Y_STATE_enum` **get_state()**

Returns :

either `Y_STATE_A` or `Y_STATE_B`, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns `Y_STATE_INVALID`.

watchdog→get_stateAtPowerOn()

YWatchdog

watchdog→stateAtPowerOn() watchdog→

get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

[Y_STATEATPOWERON_enum](#) **get_stateAtPowerOn()**

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

watchdog→get_triggerDelay()**YWatchdog****watchdog→triggerDelay()****watchdog→****get_triggerDelay()**

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

s64 **get_triggerDelay()**

Returns :

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns Y_TRIGGERDELAY_INVALID.

watchdog→**get_triggerDuration()**
watchdog→**triggerDuration()****watchdog**→
get_triggerDuration()

YWatchdog

Returns the duration of resets caused by the watchdog, in milliseconds.

s64 **get_triggerDuration()**

Returns :

an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns Y_TRIGGERDURATION_INVALID.

watchdog→get_userdata()**YWatchdog****watchdog→userdata()****watchdog→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
void * get_userdata( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

watchdog→**isOnline()****watchdog**→**isOnline()**

YWatchdog

Checks if the watchdog is currently reachable, without raising any error.

`bool isOnline()`

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

Returns :

`true` if the watchdog can be reached, and `false` otherwise

watchdog→load()**watchdog→load()****YWatchdog**

Preloads the watchdog cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

watchdog→**nextWatchdog()****watchdog**→
nextWatchdog()

YWatchdog

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

`YWatchdog * nextWatchdog()`

Returns :

a pointer to a `YWatchdog` object, corresponding to a watchdog currently online, or a `null` pointer if there are no more watchdog to enumerate.

watchdog→pulse()

YWatchdog

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
int pulse( int ms_duration)
```

Parameters :

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→**registerValueCallback()****watchdog**→
registerValueCallback()

YWatchdog

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YWatchdogValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

watchdog→**resetWatchdog()****watchdog**→
resetWatchdog()

YWatchdog

Resets the watchdog.

int resetWatchdog()

When the watchdog is running, this function must be called on a regular basis to prevent the watchdog to trigger

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_autoStart()**YWatchdog****watchdog→setAutoStart()****watchdog→****set_autoStart()**

Changes the watchdog runningstae at module power on.

```
int set_autoStart( Y_AUTOSTART_enum newval)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog runningstae at module power on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_logicalName()**YWatchdog****watchdog→setLogicalName()**
watchdog→set_logicalName()

Changes the logical name of the watchdog.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the watchdog.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

watchdog→**set_maxTimeOnStateA()****YWatchdog****watchdog**→**setMaxTimeOnStateA()****watchdog**→**set_maxTimeOnStateA()**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
int set_maxTimeOnStateA( s64 newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateB()**YWatchdog****watchdog→setMaxTimeOnStateB()**
watchdog→set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
int set_maxTimeOnStateB( s64 newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_output()**YWatchdog****watchdog→setOutput()****watchdog→set_output ()**

Changes the output state of the watchdog, when used as a simple switch (single throw).

```
int set_output( Y_OUTPUT_enum newval)
```

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_running()**YWatchdog****watchdog→setRunning()**
watchdog→set_running()

Changes the running state of the watchdog.

```
int set_running( Y_RUNNING_enum newval)
```

Parameters :

newval either Y_RUNNING_OFF or Y_RUNNING_ON, according to the running state of the watchdog

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_state()**YWatchdog****watchdog→setState()****watchdog→set_state()**

Changes the state of the watchdog (A for the idle position, B for the active position).

```
int set_state( Y_STATE_enum newval)
```

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_stateAtPowerOn()**YWatchdog****watchdog→setStateAtPowerOn()**
watchdog→set_stateAtPowerOn()

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
int set_stateAtPowerOn( Y_STATEATPOWERON_enum newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→**set_triggerDelay()****YWatchdog****watchdog**→**setTriggerDelay()****watchdog**→**set_triggerDelay()**

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

```
int set_triggerDelay( s64 newval)
```

Parameters :

newval an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDuration()**YWatchdog****watchdog→setTriggerDuration()****watchdog→****set_triggerDuration()**

Changes the duration of resets caused by the watchdog, in milliseconds.

```
int set_triggerDuration( s64 newval)
```

Parameters :

newval an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_userdata()

YWatchdog

watchdog→setUserData()**watchdog→**

set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.46. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wireless.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWireless = yoctolib.YWireless;
php	require_once('yocto_wireless.php');
c++	#include "yocto_wireless.h"
m	#import "yocto_wireless.h"
pas	uses yocto_wireless;
vb	yocto_wireless.vb
cs	yocto_wireless.cs
java	import com.yoctopuce.YoctoAPI.YWireless;
py	from yocto_wireless import *

Global functions

yFindWireless(func)

Retrieves a wireless lan interface for a given identifier.

yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

YWireless methods

wireless→adhocNetwork(ssid, securityKey)

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

wireless→describe()

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

wireless→get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

wireless→get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

wireless→get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

wireless→get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

wireless→get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

wireless→get_friendlyName()

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME . FUNCTION_NAME`.

wireless→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

wireless→get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

wireless→get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL . FUNCTIONID`.

wireless→get_linkQuality()

Returns the link quality, expressed in percent.

wireless→get_logicalName()

Returns the logical name of the wireless lan interface.

wireless→get_message()

Returns the latest status message from the wireless interface.

wireless→get_module()

Gets the YModule object for the device on which the function is located.

wireless→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

wireless→get_security()

Returns the security algorithm used by the selected wireless network.

wireless→get_ssid()

Returns the wireless network name (SSID).

wireless→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

wireless→isOnline()

Checks if the wireless lan interface is currently reachable, without raising any error.

wireless→isOnline_async(callback, context)

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

wireless→joinNetwork(ssid, securityKey)

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

wireless→load(msValidity)

Preloads the wireless lan interface cache with a specified validity duration.

wireless→load_async(msValidity, callback, context)

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

wireless→nextWireless()

Continues the enumeration of wireless lan interfaces started using yFirstWireless().

wireless→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

wireless→set_logicalName(newval)

Changes the logical name of the wireless lan interface.

wireless→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

wireless→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWireless.FindWireless() yFindWireless()yFindWireless()

YWireless

Retrieves a wireless lan interface for a given identifier.

```
YWireless* yFindWireless( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wireless lan interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWireless.isOnline()` to test if the wireless lan interface is indeed online at a given time. In case of ambiguity when looking for a wireless lan interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the wireless lan interface

Returns :

a `YWireless` object allowing you to drive the wireless lan interface.

YWireless.FirstWireless()

YWireless

yFirstWireless()**yFirstWireless()**

Starts the enumeration of wireless lan interfaces currently accessible.

YWireless* **yFirstWireless()**

Use the method `YWireless.nextWireless()` to iterate on next wireless lan interfaces.

Returns :

a pointer to a `YWireless` object, corresponding to the first wireless lan interface currently online, or a `null` pointer if there are none.

wireless→adhocNetwork()**wireless→**
adhocNetwork()

YWireless

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

```
int adhocNetwork( string ssid, string securityKey)
```

If a security key is specified, the network is protected by WEP128, since WPA is not standardized for ad-hoc networks. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid the name of the network to connect to
securityKey the network key, as a character string

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→describe()**wireless→describe()****YWireless**

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

string **describe()**

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the wireless lan interface (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wireless→**get_advertisedValue()****YWireless****wireless**→**advertisedValue()****wireless**→
get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

string **get_advertisedValue()**

Returns :

a string corresponding to the current value of the wireless lan interface (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wireless→get_channel()

YWireless

wireless→channel()**wireless→get_channel()**

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

int get_channel()

Returns :

an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns Y_CHANNEL_INVALID.

wireless→get_detectedWlans()**YWireless****wireless→detectedWlans()****wireless→**
get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

```
vector<YWlanRecord> get_detectedWlans( )
```

This list is not updated when the module is already connected to an access point (infrastructure mode). To force an update of this list, `adhocNetwork()` must be called to disconnect the module from the current network. The returned list must be unallocated by the caller.

Returns :

a list of YWlanRecord objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.

wireless→get_errorMessage()

YWireless

wireless→errorMessage()**wireless→**
get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

`string get_errorMessage()`

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the wireless lan interface object

wireless→**get_errorType()****YWireless****wireless**→**errorType()****wireless**→**get_errorType()**

Returns the numerical error code of the latest error with the wireless lan interface.

YRETCODE **get_errorType()**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the wireless lan interface object

wireless→**get_friendlyName()****YWireless****wireless**→**friendlyName()****wireless**→
get_friendlyName()

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

`string` **get_friendlyName()**

The returned string uses the logical names of the module and of the wireless lan interface if they are defined, otherwise the serial number of the module and the hardware identifier of the wireless lan interface (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the wireless lan interface using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wireless→**get_functionDescriptor()****YWireless****wireless**→**functionDescriptor()****wireless**→
get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

YFUN_DESCR **get_functionDescriptor()**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wireless→**get_functionId()**

YWireless

wireless→**functionId()****wireless**→
get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

`string get_functionId()`

For example `relay1`

Returns :

a string that identifies the wireless lan interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wireless→get_hardwareId()**YWireless****wireless→hardwareId()****wireless→**
get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

`string` **get_hardwareId()**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wireless lan interface. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the wireless lan interface (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

wireless→get_linkQuality()

YWireless

wireless→linkQuality()wireless→

get_linkQuality()

Returns the link quality, expressed in percent.

int get_linkQuality()

Returns :

an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns Y_LINKQUALITY_INVALID.

wireless→**get_logicalName()****YWireless****wireless**→**logicalName()****wireless**→
get_logicalName()

Returns the logical name of the wireless lan interface.

string **get_logicalName()**

Returns :

a string corresponding to the logical name of the wireless lan interface. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wireless→get_message()

YWireless

wireless→message()**wireless→get_message()**

Returns the latest status message from the wireless interface.

string **get_message()**

Returns :

a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns Y_MESSAGE_INVALID.

wireless→get_module()**YWireless****wireless→module()****wireless→get_module()**

Gets the YModule object for the device on which the function is located.

YModule * **get_module()**

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

wireless→get_security()**YWireless****wireless→security()****wireless→get_security()**

Returns the security algorithm used by the selected wireless network.

`Y_SECURITY_enum` **get_security()**

Returns :

a value among `Y_SECURITY_UNKNOWN`, `Y_SECURITY_OPEN`, `Y_SECURITY_WEP`, `Y_SECURITY_WPA` and `Y_SECURITY_WPA2` corresponding to the security algorithm used by the selected wireless network

On failure, throws an exception or returns `Y_SECURITY_INVALID`.

wireless→get_ssid()**YWireless****wireless→ssid()****wireless→get_ssid()**

Returns the wireless network name (SSID).

string **get_ssid()**

Returns :

a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns Y_SSID_INVALID.

wireless→get_userdata()

YWireless

wireless→userdata()**wireless→get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

`void * get_userdata()`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

wireless→**isOnline()****wireless**→**isOnline()****YWireless**

Checks if the wireless lan interface is currently reachable, without raising any error.

```
bool isOnline( )
```

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

Returns :

`true` if the wireless lan interface can be reached, and `false` otherwise

wireless→joinNetwork()**wireless→joinNetwork()****YWireless**

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

```
int joinNetwork( string ssid, string securityKey)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid the name of the network to connect to
securityKey the network key, as a character string

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→load()**wireless→load()****YWireless**

Preloads the wireless lan interface cache with a specified validity duration.

```
YRETCODE load( int msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

wireless→**nextWireless()****wireless**→
nextWireless()

YWireless

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

`YWireless *` **nextWireless()**

Returns :

a pointer to a `YWireless` object, corresponding to a wireless lan interface currently online, or a `null` pointer if there are no more wireless lan interfaces to enumerate.

wireless→registerValueCallback()**wireless→**
registerValueCallback()

YWireless

Registers the callback function that is invoked on every change of advertised value.

```
int registerValueCallback( YWirelessValueCallback callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wireless→**set_logicalName()****YWireless****wireless**→**setLogicalName()****wireless**→
set_logicalName()

Changes the logical name of the wireless lan interface.

```
int set_logicalName( const string& newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wireless lan interface.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wireless→set_userdata()**YWireless****wireless→setUserData()****wireless→**
set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
void set_userdata( void* data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

Index

A

Accelerometer 33
adhocNetwork, YWireless 1560
AnButton 75

B

Blueprint 12

C

C++ 3, 8
calibrate, YLightSensor 688
calibrateFromPoints, YAccelerometer 37
calibrateFromPoints, YCarbonDioxide 117
calibrateFromPoints, YCompass 185
calibrateFromPoints, YCurrent 225
calibrateFromPoints, YGenericSensor 500
calibrateFromPoints, YGyro 546
calibrateFromPoints, YHumidity 622
calibrateFromPoints, YLightSensor 689
calibrateFromPoints, YMagnetometer 728
calibrateFromPoints, YPower 899
calibrateFromPoints, YPressure 942
calibrateFromPoints, YQt 1042
calibrateFromPoints, YSensor 1180
calibrateFromPoints, YTemperature 1254
calibrateFromPoints, YTilt 1295
calibrateFromPoints, YVoc 1334
calibrateFromPoints, YVoltage 1373
callbackLogin, YNetwork 820
cancel3DCalibration, YRefFrame 1108
CarbonDioxide 113
CheckLogicalName, YAPI 14
clear, YDisplayLayer 412
clearConsole, YDisplayLayer 413
Clock 1077
ColorLed 152
Compass 181
Configuration 1104
consoleOut, YDisplayLayer 414
copyLayerContent, YDisplay 368
Current 221

D

Data 291, 293, 305
DataLogger 260
delayedPulse, YDigitalIO 324
delayedPulse, YRelay 1144
delayedPulse, YWatchdog 1516
describe, YAccelerometer 38
describe, YAnButton 79
describe, YCarbonDioxide 118
describe, YColorLed 155

describe, YCompass 186
describe, YCurrent 226
describe, YDataLogger 263
describe, YDigitalIO 325
describe, YDisplay 369
describe, YDualPower 446
describe, YFiles 471
describe, YGenericSensor 501
describe, YGyro 547
describe, YHubPort 596
describe, YHumidity 623
describe, YLed 660
describe, YLightSensor 690
describe, YMagnetometer 729
describe, YModule 776
describe, YNetwork 821
describe, YOsControl 875
describe, YPower 900
describe, YPressure 943
describe, YPwmOutput 981
describe, YPwmPowerSource 1018
describe, YQt 1043
describe, YRealTimeClock 1080
describe, YRefFrame 1109
describe, YRelay 1145
describe, YSensor 1181
describe, YServo 1219
describe, YTemperature 1255
describe, YTilt 1296
describe, YVoc 1335
describe, YVoltage 1374
describe, YVSource 1411
describe, YWakeUpMonitor 1444
describe, YWakeUpSchedule 1479
describe, YWatchdog 1517
describe, YWireless 1561
Digital 320
DisableExceptions, YAPI 15
Display 364
DisplayLayer 411
download, YFiles 472
download, YModule 777
drawBar, YDisplayLayer 415
drawBitmap, YDisplayLayer 416
drawCircle, YDisplayLayer 417
drawDisc, YDisplayLayer 418
drawImage, YDisplayLayer 419
drawPixel, YDisplayLayer 420
drawRect, YDisplayLayer 421
drawText, YDisplayLayer 422
dutyCycleMove, YPwmOutput 982

E

EnableExceptions, YAPI 16

Error 7
External 443

F

fade, YDisplay 370
Files 468
FindAccelerometer, YAccelerometer 35
FindAnButton, YAnButton 77
FindCarbonDioxide, YCarbonDioxide 115
FindColorLed, YColorLed 153
FindCompass, YCompass 183
FindCurrent, YCurrent 223
FindDataLogger, YDataLogger 261
FindDigitalIO, YDigitalIO 322
FindDisplay, YDisplay 366
FindDualPower, YDualPower 444
FindFiles, YFiles 469
FindGenericSensor, YGenericSensor 498
FindGyro, YGyro 544
FindHubPort, YHubPort 594
FindHumidity, YHumidity 620
FindLed, YLed 658
FindLightSensor, YLightSensor 687
FindMagnetometer, YMagnetometer 727
FindModule, YModule 775
FindNetwork, YNetwork 819
FindOsControl, YOsControl 874
FindPower, YPower 898
FindPressure, YPressure 941
FindPwmOutput, YPwmOutput 980
FindPwmPowerSource, YPwmPowerSource 1017
FindQt, YQt 1041
FindRealTimeClock, YRealTimeClock 1079
FindRefFrame, YRefFrame 1107
FindRelay, YRelay 1143
FindSensor, YSensor 1179
FindServo, YServo 1218
FindTemperature, YTemperature 1253
FindTilt, YTilt 1294
FindVoc, YVoc 1333
FindVoltage, YVoltage 1372
FindVSource, YVSource 1410
FindWakeUpMonitor, YWakeUpMonitor 1443
FindWakeUpSchedule, YWakeUpSchedule 1478
FindWatchdog, YWatchdog 1515
FindWireless, YWireless 1558
forgetAllDataStreams, YDataLogger 264
format_fs, YFiles 473
Formatted 291
Frame 1104
FreeAPI, YAPI 17
functionCount, YModule 778
functionId, YModule 779
functionName, YModule 780
Functions 13
functionValue, YModule 781

G

General 13
GenericSensor 496
get_3DCalibrationHint, YRefFrame 1110
get_3DCalibrationLogMsg, YRefFrame 1111
get_3DCalibrationProgress, YRefFrame 1112
get_3DCalibrationStage, YRefFrame 1113
get_3DCalibrationStageProgress, YRefFrame 1114
get_adminPassword, YNetwork 822
get_advertisedValue, YAccelerometer 39
get_advertisedValue, YAnButton 80
get_advertisedValue, YCarbonDioxide 119
get_advertisedValue, YColorLed 156
get_advertisedValue, YCompass 187
get_advertisedValue, YCurrent 227
get_advertisedValue, YDataLogger 265
get_advertisedValue, YDigitalIO 326
get_advertisedValue, YDisplay 371

get_advertisedValue, YDualPower 447
 get_advertisedValue, YFiles 474
 get_advertisedValue, YGenericSensor 502
 get_advertisedValue, YGyro 548
 get_advertisedValue, YHubPort 597
 get_advertisedValue, YHumidity 624
 get_advertisedValue, YLed 661
 get_advertisedValue, YLightSensor 691
 get_advertisedValue, YMagnetometer 730
 get_advertisedValue, YNetwork 823
 get_advertisedValue, YOsControl 876
 get_advertisedValue, YPower 901
 get_advertisedValue, YPressure 944
 get_advertisedValue, YPwmOutput 983
 get_advertisedValue, YPwmPowerSource 1019
 get_advertisedValue, YQt 1044
 get_advertisedValue, YRealTimeClock 1081
 get_advertisedValue, YRefFrame 1115
 get_advertisedValue, YRelay 1146
 get_advertisedValue, YSensor 1182
 get_advertisedValue, YServo 1220
 get_advertisedValue, YTemperature 1256
 get_advertisedValue, YTilt 1297
 get_advertisedValue, YVoc 1336
 get_advertisedValue, YVoltage 1375
 get_advertisedValue, YVSource 1412
 get_advertisedValue, YWakeUpMonitor 1445
 get_advertisedValue, YWakeUpSchedule 1480
 get_advertisedValue, YWatchdog 1518
 get_advertisedValue, YWireless 1562
 get_analogCalibration, YAnButton 81
 get_autoStart, YDataLogger 266
 get_autoStart, YWatchdog 1519
 get_averageValue, YDataStream 306
 get_averageValue, YMeasure 766
 get_baudRate, YHubPort 598
 get_beacon, YModule 782
 get_bearing, YRefFrame 1116
 get_bitDirection, YDigitalIO 327
 get_bitOpenDrain, YDigitalIO 328
 get_bitPolarity, YDigitalIO 329
 get_bitState, YDigitalIO 330
 get_blinking, YLed 662
 get_brightness, YDisplay 372
 get_calibratedValue, YAnButton 82
 get_calibrationMax, YAnButton 83
 get_calibrationMin, YAnButton 84
 get_callbackCredentials, YNetwork 824
 get_callbackEncoding, YNetwork 825
 get_callbackMaxDelay, YNetwork 826
 get_callbackMethod, YNetwork 827
 get_callbackMinDelay, YNetwork 828
 get_callbackUrl, YNetwork 829
 get_channel, YWireless 1563
 get_columnCount, YDataStream 307
 get_columnNames, YDataStream 308
 get_cosPhi, YPower 902
 get_countdown, YRelay 1147
 get_countdown, YWatchdog 1520
 get_currentRawValue, YAccelerometer 40
 get_currentRawValue, YCarbonDioxide 120
 get_currentRawValue, YCompass 188
 get_currentRawValue, YCurrent 228
 get_currentRawValue, YGenericSensor 503
 get_currentRawValue, YGyro 549
 get_currentRawValue, YHumidity 625
 get_currentRawValue, YLightSensor 692
 get_currentRawValue, YMagnetometer 731
 get_currentRawValue, YPower 903
 get_currentRawValue, YPressure 945
 get_currentRawValue, YQt 1045
 get_currentRawValue, YSensor 1183
 get_currentRawValue, YTemperature 1257
 get_currentRawValue, YTilt 1298
 get_currentRawValue, YVoc 1337
 get_currentRawValue, YVoltage 1376
 get_currentRunIndex, YDataLogger 267
 get_currentValue, YAccelerometer 41
 get_currentValue, YCarbonDioxide 121
 get_currentValue, YCompass 189
 get_currentValue, YCurrent 229
 get_currentValue, YGenericSensor 504
 get_currentValue, YGyro 550
 get_currentValue, YHumidity 626
 get_currentValue, YLightSensor 693
 get_currentValue, YMagnetometer 732
 get_currentValue, YPower 904
 get_currentValue, YPressure 946
 get_currentValue, YQt 1046
 get_currentValue, YSensor 1184
 get_currentValue, YTemperature 1258
 get_currentValue, YTilt 1299
 get_currentValue, YVoc 1338
 get_currentValue, YVoltage 1377
 get_data, YDataStream 309
 get_dataRows, YDataStream 310
 get_dataSamplesIntervalMs, YDataStream 311
 get_dataSets, YDataLogger 268
 get_dataStreams, YDataLogger 269
 get_dateTime, YRealTimeClock 1082
 get_detectedWlans, YWireless 1564
 get_discoverable, YNetwork 830
 get_display, YDisplayLayer 423
 get_displayHeight, YDisplay 373
 get_displayHeight, YDisplayLayer 424
 get_displayLayer, YDisplay 374
 get_displayType, YDisplay 375
 get_displayWidth, YDisplay 376
 get_displayWidth, YDisplayLayer 425
 get_duration, YDataStream 312
 get_dutyCycle, YPwmOutput 984
 get_dutyCycleAtPowerOn, YPwmOutput 985
 get_enabled, YDisplay 377
 get_enabled, YHubPort 599
 get_enabled, YPwmOutput 986
 get_enabled, YServo 1221
 get_enabledAtPowerOn, YPwmOutput 987
 get_enabledAtPowerOn, YServo 1222

get_endTimeUTC, YDataSet 294
get_endTimeUTC, YMeasure 767
get_errorMessage, YAccelerometer 42
get_errorMessage, YAnButton 85
get_errorMessage, YCarbonDioxide 122
get_errorMessage, YColorLed 157
get_errorMessage, YCompass 190
get_errorMessage, YCurrent 230
get_errorMessage, YDataLogger 270
get_errorMessage, YDigitalIO 331
get_errorMessage, YDisplay 378
get_errorMessage, YDualPower 448
get_errorMessage, YFiles 475
get_errorMessage, YGenericSensor 505
get_errorMessage, YGyro 551
get_errorMessage, YHubPort 600
get_errorMessage, YHumidity 627
get_errorMessage, YLed 663
get_errorMessage, YLightSensor 694
get_errorMessage, YMagnetometer 733
get_errorMessage, YModule 783
get_errorMessage, YNetwork 831
get_errorMessage, YOsControl 877
get_errorMessage, YPower 905
get_errorMessage, YPressure 947
get_errorMessage, YPwmOutput 988
get_errorMessage, YPwmPowerSource 1020
get_errorMessage, YQt 1047
get_errorMessage, YRealTimeClock 1083
get_errorMessage, YRefFrame 1117
get_errorMessage, YRelay 1148
get_errorMessage, YSensor 1185
get_errorMessage, YServo 1223
get_errorMessage, YTemperature 1259
get_errorMessage, YTilt 1300
get_errorMessage, YVoc 1339
get_errorMessage, YVoltage 1378
get_errorMessage, YVSource 1413
get_errorMessage, YWakeUpMonitor 1446
get_errorMessage, YWakeUpSchedule 1481
get_errorMessage, YWatchdog 1521
get_errorMessage, YWireless 1565
get_errorType, YAccelerometer 43
get_errorType, YAnButton 86
get_errorType, YCarbonDioxide 123
get_errorType, YColorLed 158
get_errorType, YCompass 191
get_errorType, YCurrent 231
get_errorType, YDataLogger 271
get_errorType, YDigitalIO 332
get_errorType, YDisplay 379
get_errorType, YDualPower 449
get_errorType, YFiles 476
get_errorType, YGenericSensor 506
get_errorType, YGyro 552
get_errorType, YHubPort 601
get_errorType, YHumidity 628
get_errorType, YLed 664
get_errorType, YLightSensor 695

get_errorType, YMagnetometer 734
get_errorType, YModule 784
get_errorType, YNetwork 832
get_errorType, YOsControl 878
get_errorType, YPower 906
get_errorType, YPressure 948
get_errorType, YPwmOutput 989
get_errorType, YPwmPowerSource 1021
get_errorType, YQt 1048
get_errorType, YRealTimeClock 1084
get_errorType, YRefFrame 1118
get_errorType, YRelay 1149
get_errorType, YSensor 1186
get_errorType, YServo 1224
get_errorType, YTemperature 1260
get_errorType, YTilt 1301
get_errorType, YVoc 1340
get_errorType, YVoltage 1379
get_errorType, YVSource 1414
get_errorType, YWakeUpMonitor 1447
get_errorType, YWakeUpSchedule 1482
get_errorType, YWatchdog 1522
get_errorType, YWireless 1566
get_extPowerFailure, YVSource 1415
get_extVoltage, YDualPower 450
get_failure, YVSource 1416
get_filesCount, YFiles 477
get_firmwareRelease, YModule 785
get_freeSpace, YFiles 478
get_frequency, YPwmOutput 990
get_friendlyName, YAccelerometer 44
get_friendlyName, YAnButton 87
get_friendlyName, YCarbonDioxide 124
get_friendlyName, YColorLed 159
get_friendlyName, YCompass 192
get_friendlyName, YCurrent 232
get_friendlyName, YDataLogger 272
get_friendlyName, YDigitalIO 333
get_friendlyName, YDisplay 380
get_friendlyName, YDualPower 451
get_friendlyName, YFiles 479
get_friendlyName, YGenericSensor 507
get_friendlyName, YGyro 553
get_friendlyName, YHubPort 602
get_friendlyName, YHumidity 629
get_friendlyName, YLed 665
get_friendlyName, YLightSensor 696
get_friendlyName, YMagnetometer 735
get_friendlyName, YNetwork 833
get_friendlyName, YOsControl 879
get_friendlyName, YPower 907
get_friendlyName, YPressure 949
get_friendlyName, YPwmOutput 991
get_friendlyName, YPwmPowerSource 1022
get_friendlyName, YQt 1049
get_friendlyName, YRealTimeClock 1085
get_friendlyName, YRefFrame 1119
get_friendlyName, YRelay 1150
get_friendlyName, YSensor 1187

get_friendlyName, YServo 1225
get_friendlyName, YTemperature 1261
get_friendlyName, YTilt 1302
get_friendlyName, YVoc 1341
get_friendlyName, YVoltage 1380
get_friendlyName, YVSource 1417
get_friendlyName, YWakeUpMonitor 1448
get_friendlyName, YWakeUpSchedule 1483
get_friendlyName, YWatchdog 1523
get_friendlyName, YWireless 1567
get_functionDescriptor, YAccelerometer 45
get_functionDescriptor, YAnButton 88
get_functionDescriptor, YCarbonDioxide 125
get_functionDescriptor, YColorLed 160
get_functionDescriptor, YCompass 193
get_functionDescriptor, YCurrent 233
get_functionDescriptor, YDataLogger 273
get_functionDescriptor, YDigitalIO 334
get_functionDescriptor, YDisplay 381
get_functionDescriptor, YDualPower 452
get_functionDescriptor, YFiles 480
get_functionDescriptor, YGenericSensor 508
get_functionDescriptor, YGyro 554
get_functionDescriptor, YHubPort 603
get_functionDescriptor, YHumidity 630
get_functionDescriptor, YLed 666
get_functionDescriptor, YLightSensor 697
get_functionDescriptor, YMagnetometer 736
get_functionDescriptor, YNetwork 834
get_functionDescriptor, YOsControl 880
get_functionDescriptor, YPower 908
get_functionDescriptor, YPressure 950
get_functionDescriptor, YPwmOutput 992
get_functionDescriptor, YPwmPowerSource 1023
get_functionDescriptor, YQt 1050
get_functionDescriptor, YRealTimeClock 1086
get_functionDescriptor, YRefFrame 1120
get_functionDescriptor, YRelay 1151
get_functionDescriptor, YSensor 1188
get_functionDescriptor, YServo 1226
get_functionDescriptor, YTemperature 1262
get_functionDescriptor, YTilt 1303
get_functionDescriptor, YVoc 1342
get_functionDescriptor, YVoltage 1381
get_functionDescriptor, YVSource 1418
get_functionDescriptor, YWakeUpMonitor 1449
get_functionDescriptor, YWakeUpSchedule 1484
get_functionDescriptor, YWatchdog 1524
get_functionDescriptor, YWireless 1568
get_functionId, YAccelerometer 46
get_functionId, YAnButton 89
get_functionId, YCarbonDioxide 126
get_functionId, YColorLed 161
get_functionId, YCompass 194
get_functionId, YCurrent 234
get_functionId, YDataLogger 274
get_functionId, YDataSet 295
get_functionId, YDigitalIO 335
get_functionId, YDisplay 382

get_functionId, YDualPower 453
get_functionId, YFiles 481
get_functionId, YGenericSensor 509
get_functionId, YGyro 555
get_functionId, YHubPort 604
get_functionId, YHumidity 631
get_functionId, YLed 667
get_functionId, YLightSensor 698
get_functionId, YMagnetometer 737
get_functionId, YNetwork 835
get_functionId, YOsControl 881
get_functionId, YPower 909
get_functionId, YPressure 951
get_functionId, YPwmOutput 993
get_functionId, YPwmPowerSource 1024
get_functionId, YQt 1051
get_functionId, YRealTimeClock 1087
get_functionId, YRefFrame 1121
get_functionId, YRelay 1152
get_functionId, YSensor 1189
get_functionId, YServo 1227
get_functionId, YTemperature 1263
get_functionId, YTilt 1304
get_functionId, YVoc 1343
get_functionId, YVoltage 1382
get_functionId, YVSource 1419
get_functionId, YWakeUpMonitor 1450
get_functionId, YWakeUpSchedule 1485
get_functionId, YWatchdog 1525
get_functionId, YWireless 1569
get_hardwareId, YAccelerometer 47
get_hardwareId, YAnButton 90
get_hardwareId, YCarbonDioxide 127
get_hardwareId, YColorLed 162
get_hardwareId, YCompass 195
get_hardwareId, YCurrent 235
get_hardwareId, YDataLogger 275
get_hardwareId, YDataSet 296
get_hardwareId, YDigitalIO 336
get_hardwareId, YDisplay 383
get_hardwareId, YDualPower 454
get_hardwareId, YFiles 482
get_hardwareId, YGenericSensor 510
get_hardwareId, YGyro 556
get_hardwareId, YHubPort 605
get_hardwareId, YHumidity 632
get_hardwareId, YLed 668
get_hardwareId, YLightSensor 699
get_hardwareId, YMagnetometer 738
get_hardwareId, YModule 786
get_hardwareId, YNetwork 836
get_hardwareId, YOsControl 882
get_hardwareId, YPower 910
get_hardwareId, YPressure 952
get_hardwareId, YPwmOutput 994
get_hardwareId, YPwmPowerSource 1025
get_hardwareId, YQt 1052
get_hardwareId, YRealTimeClock 1088
get_hardwareId, YRefFrame 1122

get_hardwareId, YRelay 1153
get_hardwareId, YSensor 1190
get_hardwareId, YServo 1228
get_hardwareId, YTemperature 1264
get_hardwareId, YTilt 1305
get_hardwareId, YVoc 1344
get_hardwareId, YVoltage 1383
get_hardwareId, YVSource 1420
get_hardwareId, YWakeUpMonitor 1451
get_hardwareId, YWakeUpSchedule 1486
get_hardwareId, YWatchdog 1526
get_hardwareId, YWireless 1570
get_heading, YGyro 557
get_highestValue, YAccelerometer 48
get_highestValue, YCarbonDioxide 128
get_highestValue, YCompass 196
get_highestValue, YCurrent 236
get_highestValue, YGenericSensor 511
get_highestValue, YGyro 558
get_highestValue, YHumidity 633
get_highestValue, YLightSensor 700
get_highestValue, YMagnetometer 739
get_highestValue, YPower 911
get_highestValue, YPressure 953
get_highestValue, YQt 1053
get_highestValue, YSensor 1191
get_highestValue, YTemperature 1265
get_highestValue, YTilt 1306
get_highestValue, YVoc 1345
get_highestValue, YVoltage 1384
get_hours, YWakeUpSchedule 1487
get_hslColor, YColorLed 163
get_icon2d, YModule 787
get_ipAddress, YNetwork 837
get_isPressed, YAnButton 91
get_lastLogs, YModule 788
get_lastTimePressed, YAnButton 92
get_lastTimeReleased, YAnButton 93
get_layerCount, YDisplay 384
get_layerHeight, YDisplay 385
get_layerHeight, YDisplayLayer 426
get_layerWidth, YDisplay 386
get_layerWidth, YDisplayLayer 427
get_linkQuality, YWireless 1571
get_list, YFiles 483
get_logFrequency, YAccelerometer 49
get_logFrequency, YCarbonDioxide 129
get_logFrequency, YCompass 197
get_logFrequency, YCurrent 237
get_logFrequency, YGenericSensor 512
get_logFrequency, YGyro 559
get_logFrequency, YHumidity 634
get_logFrequency, YLightSensor 701
get_logFrequency, YMagnetometer 740
get_logFrequency, YPower 912
get_logFrequency, YPressure 954
get_logFrequency, YQt 1054
get_logFrequency, YSensor 1192
get_logFrequency, YTemperature 1266

get_logFrequency, YTilt 1307
get_logFrequency, YVoc 1346
get_logFrequency, YVoltage 1385
get_logicalName, YAccelerometer 50
get_logicalName, YAnButton 94
get_logicalName, YCarbonDioxide 130
get_logicalName, YColorLed 164
get_logicalName, YCompass 198
get_logicalName, YCurrent 238
get_logicalName, YDataLogger 276
get_logicalName, YDigitalIO 337
get_logicalName, YDisplay 387
get_logicalName, YDualPower 455
get_logicalName, YFiles 484
get_logicalName, YGenericSensor 513
get_logicalName, YGyro 560
get_logicalName, YHubPort 606
get_logicalName, YHumidity 635
get_logicalName, YLed 669
get_logicalName, YLightSensor 702
get_logicalName, YMagnetometer 741
get_logicalName, YModule 789
get_logicalName, YNetwork 838
get_logicalName, YOsControl 883
get_logicalName, YPower 913
get_logicalName, YPressure 955
get_logicalName, YPwmOutput 995
get_logicalName, YPwmPowerSource 1026
get_logicalName, YQt 1055
get_logicalName, YRealTimeClock 1089
get_logicalName, YRefFrame 1123
get_logicalName, YRelay 1154
get_logicalName, YSensor 1193
get_logicalName, YServo 1229
get_logicalName, YTemperature 1267
get_logicalName, YTilt 1308
get_logicalName, YVoc 1347
get_logicalName, YVoltage 1386
get_logicalName, YVSource 1421
get_logicalName, YWakeUpMonitor 1452
get_logicalName, YWakeUpSchedule 1488
get_logicalName, YWatchdog 1527
get_logicalName, YWireless 1572
get_lowestValue, YAccelerometer 51
get_lowestValue, YCarbonDioxide 131
get_lowestValue, YCompass 199
get_lowestValue, YCurrent 239
get_lowestValue, YGenericSensor 514
get_lowestValue, YGyro 561
get_lowestValue, YHumidity 636
get_lowestValue, YLightSensor 703
get_lowestValue, YMagnetometer 742
get_lowestValue, YPower 914
get_lowestValue, YPressure 956
get_lowestValue, YQt 1056
get_lowestValue, YSensor 1194
get_lowestValue, YTemperature 1268
get_lowestValue, YTilt 1309
get_lowestValue, YVoc 1348

get_lowestValue, YVoltage 1387
get_luminosity, YLed 670
get_luminosity, YModule 790
get_macAddress, YNetwork 839
get_magneticHeading, YCompass 200
get_maxTimeOnStateA, YRelay 1155
get_maxTimeOnStateA, YWatchdog 1528
get_maxTimeOnStateB, YRelay 1156
get_maxTimeOnStateB, YWatchdog 1529
get_maxValue, YDataStream 313
get_maxValue, YMeasure 768
get_measures, YDataSet 297
get_message, YWireless 1573
get_meter, YPower 915
get_meterTimer, YPower 916
get_minutes, YWakeUpSchedule 1489
get_minutesA, YWakeUpSchedule 1490
get_minutesB, YWakeUpSchedule 1491
get_minValue, YDataStream 314
get_minValue, YMeasure 769
get_module, YAccelerometer 52
get_module, YAnButton 95
get_module, YCarbonDioxide 132
get_module, YColorLed 165
get_module, YCompass 201
get_module, YCurrent 240
get_module, YDataLogger 277
get_module, YDigitalIO 338
get_module, YDisplay 388
get_module, YDualPower 456
get_module, YFiles 485
get_module, YGenericSensor 515
get_module, YGyro 562
get_module, YHubPort 607
get_module, YHumidity 637
get_module, YLed 671
get_module, YLightSensor 704
get_module, YMagnetometer 743
get_module, YNetwork 840
get_module, YOsControl 884
get_module, YPower 917
get_module, YPressure 957
get_module, YPwmOutput 996
get_module, YPwmPowerSource 1027
get_module, YQt 1057
get_module, YRealTimeClock 1090
get_module, YRefFrame 1124
get_module, YRelay 1157
get_module, YSensor 1195
get_module, YServo 1230
get_module, YTemperature 1269
get_module, YTilt 1310
get_module, YVoc 1349
get_module, YVoltage 1388
get_module, YVSource 1422
get_module, YWakeUpMonitor 1453
get_module, YWakeUpSchedule 1492
get_module, YWatchdog 1530
get_module, YWireless 1574

get_monthDays, YWakeUpSchedule 1493
get_months, YWakeUpSchedule 1494
get_mountOrientation, YRefFrame 1125
get_mountPosition, YRefFrame 1126
get_neutral, YServo 1231
get_nextOccurence, YWakeUpSchedule 1495
get_nextWakeUp, YWakeUpMonitor 1454
get_orientation, YDisplay 389
get_output, YRelay 1158
get_output, YWatchdog 1531
get_outputVoltage, YDigitalIO 339
get_overCurrent, YVSource 1423
get_overHeat, YVSource 1424
get_overLoad, YVSource 1425
get_period, YPwmOutput 997
get_persistentSettings, YModule 791
get_pitch, YGyro 563
get_poeCurrent, YNetwork 841
get_portDirection, YDigitalIO 340
get_portOpenDrain, YDigitalIO 341
get_portPolarity, YDigitalIO 342
get_portSize, YDigitalIO 343
get_portState, YDigitalIO 344
get_portState, YHubPort 608
get_position, YServo 1232
get_positionAtPowerOn, YServo 1233
get_power, YLed 672
get_powerControl, YDualPower 457
get_powerDuration, YWakeUpMonitor 1455
get_powerMode, YPwmPowerSource 1028
get_powerState, YDualPower 458
get_preview, YDataSet 298
get_primaryDNS, YNetwork 842
get_productId, YModule 792
get_productName, YModule 793
get_productRelease, YModule 794
get_progress, YDataSet 299
get_pulseCounter, YAnButton 96
get_pulseDuration, YPwmOutput 998
get_pulseTimer, YAnButton 97
get_pulseTimer, YRelay 1159
get_pulseTimer, YWatchdog 1532
get_quaternionW, YGyro 564
get_quaternionX, YGyro 565
get_quaternionY, YGyro 566
get_quaternionZ, YGyro 567
get_range, YServo 1234
get_rawValue, YAnButton 98
get_readiness, YNetwork 843
get_rebootCountdown, YModule 795
get_recordedData, YAccelerometer 53
get_recordedData, YCarbonDioxide 133
get_recordedData, YCompass 202
get_recordedData, YCurrent 241
get_recordedData, YGenericSensor 516
get_recordedData, YGyro 568
get_recordedData, YHumidity 638
get_recordedData, YLightSensor 705
get_recordedData, YMagnetometer 744

get_recordedData, YPower 918
 get_recordedData, YPressure 958
 get_recordedData, YQt 1058
 get_recordedData, YSensor 1196
 get_recordedData, YTemperature 1270
 get_recordedData, YTilt 1311
 get_recordedData, YVoc 1350
 get_recordedData, YVoltage 1389
 get_recording, YDataLogger 278
 get_regulationFailure, YVSource 1426
 get_reportFrequency, YAccelerometer 54
 get_reportFrequency, YCarbonDioxide 134
 get_reportFrequency, YCompass 203
 get_reportFrequency, YCurrent 242
 get_reportFrequency, YGenericSensor 517
 get_reportFrequency, YGyro 569
 get_reportFrequency, YHumidity 639
 get_reportFrequency, YLightSensor 706
 get_reportFrequency, YMagnetometer 745
 get_reportFrequency, YPower 919
 get_reportFrequency, YPressure 959
 get_reportFrequency, YQt 1059
 get_reportFrequency, YSensor 1197
 get_reportFrequency, YTemperature 1271
 get_reportFrequency, YTilt 1312
 get_reportFrequency, YVoc 1351
 get_reportFrequency, YVoltage 1390
 get_resolution, YAccelerometer 55
 get_resolution, YCarbonDioxide 135
 get_resolution, YCompass 204
 get_resolution, YCurrent 243
 get_resolution, YGenericSensor 518
 get_resolution, YGyro 570
 get_resolution, YHumidity 640
 get_resolution, YLightSensor 707
 get_resolution, YMagnetometer 746
 get_resolution, YPower 920
 get_resolution, YPressure 960
 get_resolution, YQt 1060
 get_resolution, YSensor 1198
 get_resolution, YTemperature 1272
 get_resolution, YTilt 1313
 get_resolution, YVoc 1352
 get_resolution, YVoltage 1391
 get_rgbColor, YColorLed 166
 get_rgbColorAtPowerOn, YColorLed 167
 get_roll, YGyro 571
 get_router, YNetwork 844
 get_rowCount, YDataStream 315
 get_runIndex, YDataStream 316
 get_running, YWatchdog 1533
 get_secondaryDNS, YNetwork 845
 get_security, YWireless 1575
 get_sensitivity, YAnButton 99
 get_sensorType, YTemperature 1273
 get_serialNumber, YModule 796
 get_shutdownCountdown, YOsControl 885
 get_signalRange, YGenericSensor 519
 get_signalUnit, YGenericSensor 520
 get_signalValue, YGenericSensor 521
 get_sleepCountdown, YWakeUpMonitor 1456
 get_ssid, YWireless 1576
 get_startTime, YDataStream 317
 get_startTimeUTC, YDataRun 291
 get_startTimeUTC, YDataSet 300
 get_startTimeUTC, YDataStream 318
 get_startTimeUTC, YMeasure 770
 get_startupSeq, YDisplay 390
 get_state, YRelay 1160
 get_state, YWatchdog 1534
 get_stateAtPowerOn, YRelay 1161
 get_stateAtPowerOn, YWatchdog 1535
 get_subnetMask, YNetwork 846
 get_summary, YDataSet 301
 get_timeSet, YRealTimeClock 1091
 get_timeUTC, YDataLogger 279
 get_triggerDelay, YWatchdog 1536
 get_triggerDuration, YWatchdog 1537
 get_unit, YAccelerometer 56
 get_unit, YCarbonDioxide 136
 get_unit, YCompass 205
 get_unit, YCurrent 244
 get_unit, YDataSet 302
 get_unit, YGenericSensor 522
 get_unit, YGyro 572
 get_unit, YHumidity 641
 get_unit, YLightSensor 708
 get_unit, YMagnetometer 747
 get_unit, YPower 921
 get_unit, YPressure 961
 get_unit, YQt 1061
 get_unit, YSensor 1199
 get_unit, YTemperature 1274
 get_unit, YTilt 1314
 get_unit, YVoc 1353
 get_unit, YVoltage 1392
 get_unit, YVSource 1427
 get_unixTime, YRealTimeClock 1092
 get_upTime, YModule 797
 get_usbBandwidth, YModule 798
 get_usbCurrent, YModule 799
 get_userData, YAccelerometer 57
 get_userData, YAnButton 100
 get_userData, YCarbonDioxide 137
 get_userData, YColorLed 168
 get_userData, YCompass 206
 get_userData, YCurrent 245
 get_userData, YDataLogger 280
 get_userData, YDigitalIO 345
 get_userData, YDisplay 391
 get_userData, YDualPower 459
 get_userData, YFiles 486
 get_userData, YGenericSensor 523
 get_userData, YGyro 573
 get_userData, YHubPort 609
 get_userData, YHumidity 642
 get_userData, YLed 673
 get_userData, YLightSensor 709

- get_userdata, YMagnetometer 748
- get_userdata, YModule 800
- get_userdata, YNetwork 847
- get_userdata, YOsControl 886
- get_userdata, YPower 922
- get_userdata, YPressure 962
- get_userdata, YPwmOutput 999
- get_userdata, YPwmPowerSource 1029
- get_userdata, YQt 1062
- get_userdata, YRealTimeClock 1093
- get_userdata, YRefFrame 1127
- get_userdata, YRelay 1162
- get_userdata, YSensor 1200
- get_userdata, YServo 1235
- get_userdata, YTemperature 1275
- get_userdata, YTilt 1315
- get_userdata, YVoc 1354
- get_userdata, YVoltage 1393
- get_userdata, YVSource 1428
- get_userdata, YWakeUpMonitor 1457
- get_userdata, YWakeUpSchedule 1496
- get_userdata, YWatchdog 1538
- get_userdata, YWireless 1577
- get_userPassword, YNetwork 848
- get_utcOffset, YRealTimeClock 1094
- get_valueRange, YGenericSensor 524
- get_voltage, YVSource 1429
- get_wakeUpReason, YWakeUpMonitor 1458
- get_wakeUpState, YWakeUpMonitor 1459
- get_weekDays, YWakeUpSchedule 1497
- get_wwwWatchdogDelay, YNetwork 849
- get_xValue, YAccelerometer 58
- get_xValue, YGyro 574
- get_xValue, YMagnetometer 749
- get_yValue, YAccelerometer 59
- get_yValue, YGyro 575
- get_yValue, YMagnetometer 750
- get_zValue, YAccelerometer 60
- get_zValue, YGyro 576
- get_zValue, YMagnetometer 751
- GetAPIVersion, YAPI 18
- GetTickCount, YAPI 19
- Gyroscope 542

H

- HandleEvents, YAPI 20
- hide, YDisplayLayer 428
- hslMove, YColorLed 169
- Humidity 618

I

- InitAPI, YAPI 21
- Integration 8
- Interface 33, 75, 113, 152, 181, 221, 260, 320, 364, 411, 443, 468, 496, 542, 593, 618, 657, 684, 724, 772, 815, 895, 938, 977, 1015, 1038, 1077, 1140, 1176, 1215, 1250, 1291, 1330, 1369, 1408, 1440, 1475, 1512, 1557

- Introduction 1
- isOnline, YAccelerometer 61
- isOnline, YAnButton 101
- isOnline, YCarbonDioxide 138
- isOnline, YColorLed 170
- isOnline, YCompass 207
- isOnline, YCurrent 246
- isOnline, YDataLogger 281
- isOnline, YDigitalIO 346
- isOnline, YDisplay 392
- isOnline, YDualPower 460
- isOnline, YFiles 487
- isOnline, YGenericSensor 525
- isOnline, YGyro 577
- isOnline, YHubPort 610
- isOnline, YHumidity 643
- isOnline, YLed 674
- isOnline, YLightSensor 710
- isOnline, YMagnetometer 752
- isOnline, YModule 801
- isOnline, YNetwork 850
- isOnline, YOsControl 887
- isOnline, YPower 923
- isOnline, YPressure 963
- isOnline, YPwmOutput 1000
- isOnline, YPwmPowerSource 1030
- isOnline, YQt 1063
- isOnline, YRealTimeClock 1095
- isOnline, YRefFrame 1128
- isOnline, YRelay 1163
- isOnline, YSensor 1201
- isOnline, YServo 1236
- isOnline, YTemperature 1276
- isOnline, YTilt 1316
- isOnline, YVoc 1355
- isOnline, YVoltage 1394
- isOnline, YVSource 1430
- isOnline, YWakeUpMonitor 1460
- isOnline, YWakeUpSchedule 1498
- isOnline, YWatchdog 1539
- isOnline, YWireless 1578

J

- joinNetwork, YWireless 1579

L

- Library 8
- LightSensor 684
- lineTo, YDisplayLayer 429
- load, YAccelerometer 62
- load, YAnButton 102
- load, YCarbonDioxide 139
- load, YColorLed 171
- load, YCompass 208
- load, YCurrent 247
- load, YDataLogger 282
- load, YDigitalIO 347
- load, YDisplay 393

- load, YDualPower 461
- load, YFiles 488
- load, YGenericSensor 526
- load, YGyro 578
- load, YHubPort 611
- load, YHumidity 644
- load, YLed 675
- load, YLightSensor 711
- load, YMagnetometer 753
- load, YModule 802
- load, YNetwork 851
- load, YOsControl 888
- load, YPower 924
- load, YPressure 964
- load, YPwmOutput 1001
- load, YPwmPowerSource 1031
- load, YQt 1064
- load, YRealTimeClock 1096
- load, YRefFrame 1129
- load, YRelay 1164
- load, YSensor 1202
- load, YServo 1237
- load, YTemperature 1277
- load, YTilt 1317
- load, YVoc 1356
- load, YVoltage 1395
- load, YVSource 1431
- load, YWakeUpMonitor 1461
- load, YWakeUpSchedule 1499
- load, YWatchdog 1540
- load, YWireless 1580
- loadCalibrationPoints, YAccelerometer 63
- loadCalibrationPoints, YCarbonDioxide 140
- loadCalibrationPoints, YCompass 209
- loadCalibrationPoints, YCurrent 248
- loadCalibrationPoints, YGenericSensor 527
- loadCalibrationPoints, YGyro 579
- loadCalibrationPoints, YHumidity 645
- loadCalibrationPoints, YLightSensor 712
- loadCalibrationPoints, YMagnetometer 754
- loadCalibrationPoints, YPower 925
- loadCalibrationPoints, YPressure 965
- loadCalibrationPoints, YQt 1065
- loadCalibrationPoints, YSensor 1203
- loadCalibrationPoints, YTemperature 1278
- loadCalibrationPoints, YTilt 1318
- loadCalibrationPoints, YVoc 1357
- loadCalibrationPoints, YVoltage 1396
- loadMore, YDataSet 303

M

- Magnetometer 724
- Measured 766
- Module 5, 772
- more3DCalibration, YRefFrame 1130
- move, YServo 1238
- moveTo, YDisplayLayer 430

N

- Network 815
- newSequence, YDisplay 394
- nextAccelerometer, YAccelerometer 64
- nextAnButton, YAnButton 103
- nextCarbonDioxide, YCarbonDioxide 141
- nextColorLed, YColorLed 172
- nextCompass, YCompass 210
- nextCurrent, YCurrent 249
- nextDataLogger, YDataLogger 283
- nextDigitalIO, YDigitalIO 348
- nextDisplay, YDisplay 395
- nextDualPower, YDualPower 462
- nextFiles, YFiles 489
- nextGenericSensor, YGenericSensor 528
- nextGyro, YGyro 580
- nextHubPort, YHubPort 612
- nextHumidity, YHumidity 646
- nextLed, YLed 676
- nextLightSensor, YLightSensor 713
- nextMagnetometer, YMagnetometer 755
- nextModule, YModule 803
- nextNetwork, YNetwork 852
- nextOsControl, YOsControl 889
- nextPower, YPower 926
- nextPressure, YPressure 966
- nextPwmOutput, YPwmOutput 1002
- nextPwmPowerSource, YPwmPowerSource 1032
- nextQt, YQt 1066
- nextRealTimeClock, YRealTimeClock 1097
- nextRefFrame, YRefFrame 1131
- nextRelay, YRelay 1165
- nextSensor, YSensor 1204
- nextServo, YServo 1239
- nextTemperature, YTemperature 1279
- nextTilt, YTilt 1319
- nextVoc, YVoc 1358
- nextVoltage, YVoltage 1397
- nextVSource, YVSource 1432
- nextWakeUpMonitor, YWakeUpMonitor 1462
- nextWakeUpSchedule, YWakeUpSchedule 1500
- nextWatchdog, YWatchdog 1541
- nextWireless, YWireless 1581

O

- Object 411

P

- pauseSequence, YDisplay 396
- ping, YNetwork 853
- playSequence, YDisplay 397
- Port 593
- Power 443, 895
- PreregisterHub, YAPI 22
- Pressure 938

pulse, YDigitalIO 349
pulse, YRelay 1166
pulse, YVSource 1433
pulse, YWatchdog 1542
pulseDurationMove, YPwmOutput 1003
PwmPowerSource 1015

Q

Quaternion 1038

R

Real 1077
reboot, YModule 804
Recorded 293
Reference 12, 1104
registerAnglesCallback, YGyro 581
RegisterDeviceArrivalCallback, YAPI 23
RegisterDeviceRemovalCallback, YAPI 24
RegisterHub, YAPI 25
RegisterHubDiscoveryCallback, YAPI 26
registerLogCallback, YModule 805
RegisterLogFunction, YAPI 27
registerQuaternionCallback, YGyro 582
registerTimedReportCallback, YAccelerometer 65
registerTimedReportCallback, YCarbonDioxide 142
registerTimedReportCallback, YCompass 211
registerTimedReportCallback, YCurrent 250
registerTimedReportCallback, YGenericSensor 529
registerTimedReportCallback, YGyro 583
registerTimedReportCallback, YHumidity 647
registerTimedReportCallback, YLightSensor 714
registerTimedReportCallback, YMagnetometer 756
registerTimedReportCallback, YPower 927
registerTimedReportCallback, YPressure 967
registerTimedReportCallback, YQt 1067
registerTimedReportCallback, YSensor 1205
registerTimedReportCallback, YTemperature 1280
registerTimedReportCallback, YTilt 1320
registerTimedReportCallback, YVoc 1359
registerTimedReportCallback, YVoltage 1398
registerValueCallback, YAccelerometer 66
registerValueCallback, YAnButton 104
registerValueCallback, YCarbonDioxide 143
registerValueCallback, YColorLed 173
registerValueCallback, YCompass 212
registerValueCallback, YCurrent 251
registerValueCallback, YDataLogger 284
registerValueCallback, YDigitalIO 350
registerValueCallback, YDisplay 398
registerValueCallback, YDualPower 463
registerValueCallback, YFiles 490
registerValueCallback, YGenericSensor 530
registerValueCallback, YGyro 584

registerValueCallback, YHubPort 613
registerValueCallback, YHumidity 648
registerValueCallback, YLed 677
registerValueCallback, YLightSensor 715
registerValueCallback, YMagnetometer 757
registerValueCallback, YNetwork 854
registerValueCallback, YOsControl 890
registerValueCallback, YPower 928
registerValueCallback, YPressure 968
registerValueCallback, YPwmOutput 1004
registerValueCallback, YPwmPowerSource 1033
registerValueCallback, YQt 1068
registerValueCallback, YRealTimeClock 1098
registerValueCallback, YRefFrame 1132
registerValueCallback, YRelay 1167
registerValueCallback, YSensor 1206
registerValueCallback, YServo 1240
registerValueCallback, YTemperature 1281
registerValueCallback, YTilt 1321
registerValueCallback, YVoc 1360
registerValueCallback, YVoltage 1399
registerValueCallback, YVSource 1434
registerValueCallback, YWakeUpMonitor 1463
registerValueCallback, YWakeUpSchedule 1501
registerValueCallback, YWatchdog 1543
registerValueCallback, YWireless 1582
Relay 1140
remove, YFiles 491
reset, YDisplayLayer 431
reset, YPower 929
resetAll, YDisplay 399
resetCounter, YAnButton 105
resetSleepCountDown, YWakeUpMonitor 1464
resetWatchdog, YWatchdog 1544
revertFromFlash, YModule 806
rgbMove, YColorLed 174

S

save3DCalibration, YRefFrame 1133
saveSequence, YDisplay 400
saveToFlash, YModule 807
selectColorPen, YDisplayLayer 432
selectEraser, YDisplayLayer 433
selectFont, YDisplayLayer 434
selectGrayPen, YDisplayLayer 435
Sensor 1176
Sequence 291, 293, 305
Servo 1215
set_adminPassword, YNetwork 855
set_analogCalibration, YAnButton 106
set_autoStart, YDataLogger 285
set_autoStart, YWatchdog 1545
set_beacon, YModule 808
set_bearing, YRefFrame 1134
set_bitDirection, YDigitalIO 351
set_bitOpenDrain, YDigitalIO 352
set_bitPolarity, YDigitalIO 353
set_bitState, YDigitalIO 354
set_blinking, YLed 678

set_brightness, YDisplay 401
 set_calibrationMax, YAnButton 107
 set_calibrationMin, YAnButton 108
 set_callbackCredentials, YNetwork 856
 set_callbackEncoding, YNetwork 857
 set_callbackMaxDelay, YNetwork 858
 set_callbackMethod, YNetwork 859
 set_callbackMinDelay, YNetwork 860
 set_callbackUrl, YNetwork 861
 set_discoverable, YNetwork 862
 set_dutyCycle, YPwmOutput 1005
 set_dutyCycleAtPowerOn, YPwmOutput 1006
 set_enabled, YDisplay 402
 set_enabled, YHubPort 614
 set_enabled, YPwmOutput 1007
 set_enabled, YServo 1241
 set_enabledAtPowerOn, YPwmOutput 1008
 set_enabledAtPowerOn, YServo 1242
 set_frequency, YPwmOutput 1009
 set_highestValue, YAccelerometer 67
 set_highestValue, YCarbonDioxide 144
 set_highestValue, YCompass 213
 set_highestValue, YCurrent 252
 set_highestValue, YGenericSensor 531
 set_highestValue, YGyro 585
 set_highestValue, YHumidity 649
 set_highestValue, YLightSensor 716
 set_highestValue, YMagnetometer 758
 set_highestValue, YPower 930
 set_highestValue, YPressure 969
 set_highestValue, YQt 1069
 set_highestValue, YSensor 1207
 set_highestValue, YTemperature 1282
 set_highestValue, YTilt 1322
 set_highestValue, YVoc 1361
 set_highestValue, YVoltage 1400
 set_hours, YWakeUpSchedule 1502
 set_hslColor, YColorLed 175
 set_logFrequency, YAccelerometer 68
 set_logFrequency, YCarbonDioxide 145
 set_logFrequency, YCompass 214
 set_logFrequency, YCurrent 253
 set_logFrequency, YGenericSensor 532
 set_logFrequency, YGyro 586
 set_logFrequency, YHumidity 650
 set_logFrequency, YLightSensor 717
 set_logFrequency, YMagnetometer 759
 set_logFrequency, YPower 931
 set_logFrequency, YPressure 970
 set_logFrequency, YQt 1070
 set_logFrequency, YSensor 1208
 set_logFrequency, YTemperature 1283
 set_logFrequency, YTilt 1323
 set_logFrequency, YVoc 1362
 set_logFrequency, YVoltage 1401
 set_logicalName, YAccelerometer 69
 set_logicalName, YAnButton 109
 set_logicalName, YCarbonDioxide 146
 set_logicalName, YColorLed 176
 set_logicalName, YCompass 215
 set_logicalName, YCurrent 254
 set_logicalName, YDataLogger 286
 set_logicalName, YDigitalIO 355
 set_logicalName, YDisplay 403
 set_logicalName, YDualPower 464
 set_logicalName, YFiles 492
 set_logicalName, YGenericSensor 533
 set_logicalName, YGyro 587
 set_logicalName, YHubPort 615
 set_logicalName, YHumidity 651
 set_logicalName, YLed 679
 set_logicalName, YLightSensor 718
 set_logicalName, YMagnetometer 760
 set_logicalName, YModule 809
 set_logicalName, YNetwork 863
 set_logicalName, YOsControl 891
 set_logicalName, YPower 932
 set_logicalName, YPressure 971
 set_logicalName, YPwmOutput 1010
 set_logicalName, YPwmPowerSource 1034
 set_logicalName, YQt 1071
 set_logicalName, YRealTimeClock 1099
 set_logicalName, YRefFrame 1135
 set_logicalName, YRelay 1168
 set_logicalName, YSensor 1209
 set_logicalName, YServo 1243
 set_logicalName, YTemperature 1284
 set_logicalName, YTilt 1324
 set_logicalName, YVoc 1363
 set_logicalName, YVoltage 1402
 set_logicalName, YVSource 1435
 set_logicalName, YWakeUpMonitor 1465
 set_logicalName, YWakeUpSchedule 1503
 set_logicalName, YWatchdog 1546
 set_logicalName, YWireless 1583
 set_lowestValue, YAccelerometer 70
 set_lowestValue, YCarbonDioxide 147
 set_lowestValue, YCompass 216
 set_lowestValue, YCurrent 255
 set_lowestValue, YGenericSensor 534
 set_lowestValue, YGyro 588
 set_lowestValue, YHumidity 652
 set_lowestValue, YLightSensor 719
 set_lowestValue, YMagnetometer 761
 set_lowestValue, YPower 933
 set_lowestValue, YPressure 972
 set_lowestValue, YQt 1072
 set_lowestValue, YSensor 1210
 set_lowestValue, YTemperature 1285
 set_lowestValue, YTilt 1325
 set_lowestValue, YVoc 1364
 set_lowestValue, YVoltage 1403
 set_luminosity, YLed 680
 set_luminosity, YModule 810
 set_maxTimeOnStateA, YRelay 1169
 set_maxTimeOnStateA, YWatchdog 1547
 set_maxTimeOnStateB, YRelay 1170
 set_maxTimeOnStateB, YWatchdog 1548

set_minutes, YWakeUpSchedule 1504
set_minutesA, YWakeUpSchedule 1505
set_minutesB, YWakeUpSchedule 1506
set_monthDays, YWakeUpSchedule 1507
set_months, YWakeUpSchedule 1508
set_mountPosition, YRefFrame 1136
set_neutral, YServo 1244
set_nextWakeUp, YWakeUpMonitor 1466
set_orientation, YDisplay 404
set_output, YRelay 1171
set_output, YWatchdog 1549
set_outputVoltage, YDigitalIO 356
set_period, YPwmOutput 1011
set_portDirection, YDigitalIO 357
set_portOpenDrain, YDigitalIO 358
set_portPolarity, YDigitalIO 359
set_portState, YDigitalIO 360
set_position, YServo 1245
set_positionAtPowerOn, YServo 1246
set_power, YLed 681
set_powerControl, YDualPower 465
set_powerDuration, YWakeUpMonitor 1467
set_powerMode, YPwmPowerSource 1035
set_primaryDNS, YNetwork 864
set_pulseDuration, YPwmOutput 1012
set_range, YServo 1247
set_recording, YDataLogger 287
set_reportFrequency, YAccelerometer 71
set_reportFrequency, YCarbonDioxide 148
set_reportFrequency, YCompass 217
set_reportFrequency, YCurrent 256
set_reportFrequency, YGenericSensor 535
set_reportFrequency, YGyro 589
set_reportFrequency, YHumidity 653
set_reportFrequency, YLightSensor 720
set_reportFrequency, YMagnetometer 762
set_reportFrequency, YPower 934
set_reportFrequency, YPressure 973
set_reportFrequency, YQt 1073
set_reportFrequency, YSensor 1211
set_reportFrequency, YTemperature 1286
set_reportFrequency, YTilt 1326
set_reportFrequency, YVoc 1365
set_reportFrequency, YVoltage 1404
set_resolution, YAccelerometer 72
set_resolution, YCarbonDioxide 149
set_resolution, YCompass 218
set_resolution, YCurrent 257
set_resolution, YGenericSensor 536
set_resolution, YGyro 590
set_resolution, YHumidity 654
set_resolution, YLightSensor 721
set_resolution, YMagnetometer 763
set_resolution, YPower 935
set_resolution, YPressure 974
set_resolution, YQt 1074
set_resolution, YSensor 1212
set_resolution, YTemperature 1287
set_resolution, YTilt 1327

set_resolution, YVoc 1366
set_resolution, YVoltage 1405
set_rgbColor, YColorLed 177
set_rgbColorAtPowerOn, YColorLed 178
set_running, YWatchdog 1550
set_secondaryDNS, YNetwork 865
set_sensitivity, YAnButton 110
set_sensorType, YTemperature 1288
set_signalRange, YGenericSensor 537
set_sleepCountdown, YWakeUpMonitor 1468
set_startupSeq, YDisplay 405
set_state, YRelay 1172
set_state, YWatchdog 1551
set_stateAtPowerOn, YRelay 1173
set_stateAtPowerOn, YWatchdog 1552
set_timeUTC, YDataLogger 288
set_triggerDelay, YWatchdog 1553
set_triggerDuration, YWatchdog 1554
set_unit, YGenericSensor 538
set_unixTime, YRealTimeClock 1100
set_usbBandwidth, YModule 811
set_userData, YAccelerometer 73
set_userData, YAnButton 111
set_userData, YCarbonDioxide 150
set_userData, YColorLed 179
set_userData, YCompass 219
set_userData, YCurrent 258
set_userData, YDataLogger 289
set_userData, YDigitalIO 361
set_userData, YDisplay 406
set_userData, YDualPower 466
set_userData, YFiles 493
set_userData, YGenericSensor 539
set_userData, YGyro 591
set_userData, YHubPort 616
set_userData, YHumidity 655
set_userData, YLed 682
set_userData, YLightSensor 722
set_userData, YMagnetometer 764
set_userData, YModule 812
set_userData, YNetwork 866
set_userData, YOsControl 892
set_userData, YPower 936
set_userData, YPressure 975
set_userData, YPwmOutput 1013
set_userData, YPwmPowerSource 1036
set_userData, YQt 1075
set_userData, YRealTimeClock 1101
set_userData, YRefFrame 1137
set_userData, YRelay 1174
set_userData, YSensor 1213
set_userData, YServo 1248
set_userData, YTemperature 1289
set_userData, YTilt 1328
set_userData, YVoc 1367
set_userData, YVoltage 1406
set_userData, YVSource 1436
set_userData, YWakeUpMonitor 1469
set_userData, YWakeUpSchedule 1509

set_userdata, YWatchdog 1555
set_userdata, YWireless 1584
set_userPassword, YNetwork 867
set_utcOffset, YRealTimeClock 1102
set_valueRange, YGenericSensor 540
set_voltage, YVSource 1437
set_weekDays, YWakeUpSchedule 1510
set_wwwWatchdogDelay, YNetwork 868
setAntialiasingMode, YDisplayLayer 436
setConsoleBackground, YDisplayLayer 437
setConsoleMargins, YDisplayLayer 438
setConsoleWordWrap, YDisplayLayer 439
setLayerPosition, YDisplayLayer 440
shutdown, YOsControl 893
Sleep, YAPI 28
sleep, YWakeUpMonitor 1470
sleepFor, YWakeUpMonitor 1471
sleepUntil, YWakeUpMonitor 1472
Source 1408
start3DCalibration, YRefFrame 1138
stopSequence, YDisplay 407
Supply 443
swapLayerContent, YDisplay 408

T

Temperature 1250
Tilt 1291
Time 1077
toggle_bitState, YDigitalIO 362
triggerFirmwareUpdate, YModule 813
TriggerHubDiscovery, YAPI 29

U

Unformatted 305
unhide, YDisplayLayer 441
UnregisterHub, YAPI 30
UpdateDeviceList, YAPI 31
upload, YDisplay 409
upload, YFiles 494
useDHCP, YNetwork 869
useStaticIP, YNetwork 870

V

Value 766
Variants 8
Voltage 1369, 1408
voltageMove, YVSource 1438

W

wakeUp, YWakeUpMonitor 1473
WakeUpMonitor 1440
WakeUpSchedule 1475
Watchdog 1512
Wireless 1557

Y

YAccelerometer 35-73
YAnButton 77-111
YAPI 14-31
YCarbonDioxide 115-150
yCheckLogicalName 14
YColorLed 153-179
YCompass 183-219
YCurrent 223-258
YDataLogger 261-289
YDataRun 291
YDataSet 294-303
YDataStream 306-318
YDigitalIO 322-362
yDisableExceptions 15
YDisplay 366-409
YDisplayLayer 412-441
YDualPower 444-466
yEnableExceptions 16
YFiles 469-494
yFindAccelerometer 35
yFindAnButton 77
yFindCarbonDioxide 115
yFindColorLed 153
yFindCompass 183
yFindCurrent 223
yFindDataLogger 261
yFindDigitalIO 322
yFindDisplay 366
yFindDualPower 444
yFindFiles 469
yFindGenericSensor 498
yFindGyro 544
yFindHubPort 594
yFindHumidity 620
yFindLed 658
yFindLightSensor 686
yFindMagnetometer 726
yFindModule 774
yFindNetwork 818
yFindOsControl 873
yFindPower 897
yFindPressure 940
yFindPwmOutput 979
yFindPwmPowerSource 1016
yFindQt 1040
yFindRealTimeClock 1078
yFindRefFrame 1106
yFindRelay 1142
yFindSensor 1178
yFindServo 1217
yFindTemperature 1252
yFindTilt 1293
yFindVoc 1332
yFindVoltage 1371
yFindVSource 1409
yFindWakeUpMonitor 1442
yFindWakeUpSchedule 1477

yFindWatchdog 1514
yFindWireless 1558
yFirstAccelerometer 36
yFirstAnButton 78
yFirstCarbonDioxide 116
yFirstColorLed 154
yFirstCompass 184
yFirstCurrent 224
yFirstDataLogger 262
yFirstDigitalIO 323
yFirstDisplay 367
yFirstDualPower 445
yFirstFiles 470
yFirstGenericSensor 499
yFirstGyro 545
yFirstHubPort 595
yFirstHumidity 621
yFirstLed 659
yFirstLightSensor 687
yFirstMagnetometer 727
yFirstModule 775
yFirstNetwork 819
yFirstOsControl 874
yFirstPower 898
yFirstPressure 941
yFirstPwmOutput 980
yFirstPwmPowerSource 1017
yFirstQt 1041
yFirstRealTimeClock 1079
yFirstRefFrame 1107
yFirstRelay 1143
yFirstSensor 1179
yFirstServo 1218
yFirstTemperature 1253
yFirstTilt 1294
yFirstVoc 1333
yFirstVoltage 1372
yFirstVSource 1410
yFirstWakeUpMonitor 1443
yFirstWakeUpSchedule 1478
yFirstWatchdog 1515
yFirstWireless 1559
yFreeAPI 17
YGenericSensor 498-540

yGetAPIVersion 18
yGetTickCount 19
YGyro 544-591
yHandleEvents 20
YHubPort 594-616
YHumidity 620-655
yInitAPI 21
YLed 658-682
YLightSensor 686-722
YMagnetometer 726-764
YMeasure 766-770
YModule 774-813
YNetwork 818-870
Yocto-Demo 3
Yocto-hub 593
YOsControl 873-893
YPower 897-936
yPreregisterHub 22
YPressure 940-975
YPwmOutput 979-1013
YPwmPowerSource 1016-1036
YQt 1040-1075
YRealTimeClock 1078-1102
YRefFrame 1106-1138
yRegisterDeviceArrivalCallback 23
yRegisterDeviceRemovalCallback 24
yRegisterHub 25
yRegisterHubDiscoveryCallback 26
yRegisterLogFunction 27
YRelay 1142-1174
YSensor 1178-1213
YServo 1217-1248
ySleep 28
YTemperature 1252-1289
YTilt 1293-1328
yTriggerHubDiscovery 29
yUnregisterHub 30
yUpdateDeviceList 31
YVoc 1332-1367
YVoltage 1371-1406
YVSource 1409-1438
YWakeUpMonitor 1442-1473
YWakeUpSchedule 1477-1510
YWatchdog 1514-1555
YWireless 1558-1584