# Objective-C API Reference

# Table of contents

# 1. Introduction

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

# 2. Using Yocto-Demo with Objective-C

Objective-C is language of choice for programming on Mac OS X, due to its integration with the Cocoa framework. In order to use the Objective-C library, you need XCode version 4.2 (earlier versions will not work), available freely when you run Lion. If you are still under Snow Leopard, you need to be registered as Apple developer to be able to download XCode 4.2. The Yoctopuce library is ARC compatible. You can therefore implement your projects either using the traditional *retain / release* method, or using the *Automatic Reference Counting*.

Yoctopuce Objective-C libraries[1] 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 Objective-C.

You will soon notice that the Objective-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 can find on Yoctopuce blog a detailed example[2] with video shots showing how to integrate the library into your projects.

## 2.1. Control of the Led function

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/ Doc-GettingStarted-Yocto-Demo** of the Yoctopuce library.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"
#import "yocto_led.h"


static void usage(void)
{
    NSLog(@"usage: demo <serial_number>  [ on | off ]");
    NSLog(@"       demo <logical_name> [ on | off ]");
    NSLog(@"       demo any [ on | off ]                (use any discovered device)");
    exit(1);
}
```

[1] www.yoctopuce.com/EN/libraries.php
[2] www.yoctopuce.com/EN/article/new-objective-c-library-for-mac-os-x

```
int main(int argc, const char * argv[])
{
    NSError *error;
    if(argc < 3) {
        usage();
    }

    @autoreleasepool {
        NSString *target = [NSString stringWithUTF8String:argv[1]];
        NSString *on_off = [NSString stringWithUTF8String:argv[2]];
        YLed     *led;

        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if([target isEqualToString:@"any"]){
            led =  [YLed FirstLed];
        }else{
            led =  [YLed FindLed:[target stringByAppendingString:@".led"]];
        }
        if ([led isOnline]) {
            if ([on_off isEqualToString:@"on"])
                [led set_power:Y_POWER_ON];
            else
                [led set_power:Y_POWER_OFF];
        } else {
            NSLog(@"Module not connected (check identification and USB cable)\n");
        }
    }
    return 0;
}
```

There are only a few really important lines in this example. We will look at them in details.

## yocto_api.h et yocto_led.h

These two import 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.

## [YAPI RegisterHub]

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

## [Led FindLed]

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

```
YLed *led = [Led FindLed:@"YCTOPOC1-123456.led"];
YLed *led = [Led FindLed:@"YCTOPOC1-123456.MyFunction"];
YLed *led = [Led FindLed:@"MyModule.led"];
YLed *led = [Led FindLed:@"MyModule.MyFunction"];
YLed *led = [Led FindLed:@"MyFunction"];
```

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

## isOnline

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

### set_power

The `set_power()` function of the objet returned by `YLed.FindLed` allows you to turn on and off the led. The argument is YLed.POWER_ON or YLed.POWER_OFF. In the reference on the programming interface, you will find more methods to precisely control the luminosity and make the led blink automatically.

## 2.2. Control of the module part

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

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

static void usage(const char *exe)
{
    NSLog(@"usage: %s <serial or logical name> [ON/OFF]\n",exe);
    exit(1);
}


int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if(argc < 2)
            usage(argv[0]);
        NSString *serial_or_name =[NSString stringWithUTF8String:argv[1]];
        // use serial or logical name
        YModule *module = [YModule FindModule:serial_or_name];
        if ([module isOnline]) {
            if (argc > 2) {
                if (strcmp(argv[2], "ON")==0)
                    [module setBeacon:Y_BEACON_ON];
                else
                    [module setBeacon:Y_BEACON_OFF];
            }
            NSLog(@"serial:       %@\n", [module serialNumber]);
            NSLog(@"logical name: %@\n", [module logicalName]);
            NSLog(@"luminosity:   %d\n", [module luminosity]);
            NSLog(@"beacon:       ");
            if ([module beacon] == Y_BEACON_ON)
               NSLog(@"ON\n");
            else
               NSLog(@"OFF\n");
            NSLog(@"upTime:       %ld sec\n", [module upTime]/1000);
            NSLog(@"USB current:  %d mA\n",  [module usbCurrent]);
            NSLog(@"logs:  %@\n",  [module get_lastLogs]);
        } else {
            NSLog(@"%@ not connected (check identification and USB cable)\n",
                serial_or_name);
        }
    }
    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.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

static void usage(const char *exe)
{
    NSLog(@"usage: %s <serial> <newLogicalName>\n",exe);
    exit(1);
}


int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb" :&error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }

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

        NSString *serial_or_name =[NSString stringWithUTF8String:argv[1]];
        // use serial or logical name
        YModule *module = [YModule FindModule:serial_or_name];

        if (module.isOnline) {
            if (argc >= 3){
                NSString *newname =  [NSString stringWithUTF8String:argv[2]];
                if (![YAPI CheckLogicalName:newname]){
                    NSLog(@"Invalid name (%@)\n", newname);
                    usage(argv[0]);
                }
                module.logicalName = newname;
                [module saveToFlash];
            }
            NSLog(@"Current name: %@\n", module.logicalName);
        } else {
            NSLog(@"%@ not connected (check identification and USB cable)\n",
                serial_or_name);
        }
    }
    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.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

int main (int argc, const char * argv[])
{
    NSError *error;
    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb" :&error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@\n", [error localizedDescription]);
            return 1;
        }

        NSLog(@"Device list:\n");

        YModule *module = [YModule FirstModule];
        while (module != nil) {
            NSLog(@"%@ %@",module.serialNumber, module.productName);
            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 `errMessage()` methods. Their returned values contain the same information as in the exceptions when they are active.

# 3. Reference

## 3.1. General functions

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>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 |
|---|
| **yCheckLogicalName**(**name**)<br>Checks if a given string is valid as logical name for a module or a function. |
| **yDisableExceptions**()<br>Disables the use of exceptions to report runtime errors. |
| **yEnableExceptions**()<br>Re-enables the use of exceptions for runtime error handling. |
| **yEnableUSBHost**(**osContext**)<br>This function is used only on Android. |
| **yFreeAPI**()<br>Frees dynamically allocated memory blocks used by the Yoctopuce library. |
| **yGetAPIVersion**()<br>Returns the version identifier for the Yoctopuce library in use. |
| **yGetTickCount**()<br>Returns the current value of a monotone millisecond-based time counter. |
| **yHandleEvents**(**errmsg**)<br>Maintains the device-to-library communication channel. |
| **yInitAPI**(**mode**, **errmsg**)<br>Initializes the Yoctopuce programming library explicitly. |
| **yPreregisterHub**(**url**, **errmsg**)<br>Fault-tolerant alternative to RegisterHub(). |
| **yRegisterDeviceArrivalCallback**(**arrivalCallback**)<br>Register a callback function, to be called each time a device is plugged. |
| **yRegisterDeviceRemovalCallback**(**removalCallback**)<br>Register a callback function, to be called each time a device is unplugged. |
| **yRegisterHub**(**url**, **errmsg**)<br>Setup the Yoctopuce library to use modules connected on a given machine. |
| **yRegisterHubDiscoveryCallback**(**hubDiscoveryCallback**) |

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.

## 3.2. Accelerometer function interface

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

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

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

**Global functions**

**yFindAccelerometer(func)**

Retrieves an accelerometer for a given identifier.

**yFirstAccelerometer()**

Starts the enumeration of accelerometers currently accessible.

**`YAccelerometer` methods**

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

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

**accelerometer→describe()**

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

**accelerometer→get_advertisedValue()**

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

**accelerometer→get_currentRawValue()**

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

**accelerometer→get_currentValue()**

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

**accelerometer→get_errorMessage()**

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

**accelerometer→get_errorType()**

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

**accelerometer→get_friendlyName()**

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

**accelerometer→get_functionDescriptor()**

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

**accelerometer→get_functionId()**

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

**accelerometer→get_hardwareId()**

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

**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.

## 3.3. Altitude function interface

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

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

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

**Global functions**

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

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

**`YAltitude` methods**

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

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

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

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

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

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

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

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

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

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

**altitude→get_hardwareId()**
Returns the unique hardware identifier of the altimeter in the form `SERIAL.FUNCTIONID`.

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

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

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

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

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

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

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

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

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

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

**altitude→get_unit()**
Returns the measuring unit for the altitude.

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

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

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

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

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

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

**altitude→nextAltitude()**
Continues the enumeration of altimeters started using `yFirstAltitude()`.

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

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

**altitude→set_currentValue(newval)**
Changes the current estimated altitude.

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

**altitude→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**altitude→set_logicalName(newval)**

Changes the logical name of the altimeter.

**altitude→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**altitude→set_qnh(newval)**

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

**altitude→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**altitude→set_resolution(newval)**

Changes the resolution of the measured physical values.

**altitude→set_userData(data)**

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

**altitude→wait_async(callback, context)**

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

## 3.4. AnButton function interface

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

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

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

**Global functions**

**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 transitioned from open to closed).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**anbutton→set_calibrationMax(newval)**

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

**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.

## 3.5. CarbonDioxide function interface

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

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

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

**Global functions**

**yFindCarbonDioxide(func)**

Retrieves a CO2 sensor for a given identifier.

**yFirstCarbonDioxide()**

Starts the enumeration of CO2 sensors currently accessible.

**`YCarbonDioxide` methods**

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

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

**carbondioxide→describe()**

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

**carbondioxide→get_advertisedValue()**

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

**carbondioxide→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

**carbondioxide→get_currentValue()**

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

**carbondioxide→get_errorMessage()**

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

**carbondioxide→get_errorType()**

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

**carbondioxide→get_friendlyName()**

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

**carbondioxide→get_functionDescriptor()**

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

**carbondioxide→get_functionId()**

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

**carbondioxide→get_hardwareId()**

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.

## 3.6. ColorLed function interface

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

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

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

**Global functions**

**yFindColorLed(func)**

Retrieves an RGB led for a given identifier.

**yFirstColorLed()**

Starts the enumeration of RGB leds currently accessible.

**`YColorLed` methods**

**colorled→describe()**

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

**colorled→get_advertisedValue()**

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

**colorled→get_errorMessage()**

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

**colorled→get_errorType()**

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

**colorled→get_friendlyName()**

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

**colorled→get_functionDescriptor()**

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

**colorled→get_functionId()**

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

**colorled→get_hardwareId()**

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

**colorled→get_hslColor()**

Returns the current HSL color of the led.

**colorled→get_logicalName()**

Returns the logical name of the RGB led.

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

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

**colorled→get_rgbColor()**
: Returns the current RGB color of the led.

**colorled→get_rgbColorAtPowerOn()**
: Returns the configured color to be displayed when the module is turned on.

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

**colorled→hslMove(hsl_target, ms_duration)**
: Performs a smooth transition in the HSL color space between the current color and a target color.

**colorled→isOnline()**
: Checks if the RGB led is currently reachable, without raising any error.

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

**colorled→load(msValidity)**
: Preloads the RGB led cache with a specified validity duration.

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

**colorled→nextColorLed()**
: Continues the enumeration of RGB leds started using `yFirstColorLed()`.

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

**colorled→rgbMove(rgb_target, ms_duration)**
: Performs a smooth transition in the RGB color space between the current color and a target color.

**colorled→set_hslColor(newval)**
: Changes the current color of the led, using a color HSL.

**colorled→set_logicalName(newval)**
: Changes the logical name of the RGB led.

**colorled→set_rgbColor(newval)**
: Changes the current color of the led, using a RGB color.

**colorled→set_rgbColorAtPowerOn(newval)**
: Changes the color that the led will display by default when the module is turned on.

**colorled→set_userData(data)**
: Stores a user context provided as argument in the userData attribute of the function.

**colorled→wait_async(callback, context)**
: Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## 3.7. Compass function interface

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

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

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

| Global functions |
|---|
| **yFindCompass(func)**<br>Retrieves a compass for a given identifier. |
| **yFirstCompass()**<br>Starts the enumeration of compasses currently accessible. |
| **`YCompass` methods** |
| **compass→calibrateFromPoints(rawValues, refValues)**<br>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure. |
| **compass→describe()**<br>Returns a short text that describes unambiguously the instance of the compass in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **compass→get_advertisedValue()**<br>Returns the current value of the compass (no more than 6 characters). |
| **compass→get_currentRawValue()**<br>Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number. |
| **compass→get_currentValue()**<br>Returns the current value of the relative bearing, in degrees, as a floating point number. |
| **compass→get_errorMessage()**<br>Returns the error message of the latest error with the compass. |
| **compass→get_errorType()**<br>Returns the numerical error code of the latest error with the compass. |
| **compass→get_friendlyName()**<br>Returns a global identifier of the compass in the format `MODULE_NAME.FUNCTION_NAME`. |
| **compass→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **compass→get_functionId()**<br>Returns the hardware identifier of the compass, without reference to the module. |
| **compass→get_hardwareId()**<br>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.

## 3.8. Current function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_current.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YCurrent = yoctolib.YCurrent; |
| php | require_once('yocto_current.php'); |
| cpp | #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, in mA, as a floating point number.

**current→get_currentValue()**

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

**current→get_errorMessage()**

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

**current→get_errorType()**

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

**current→get_friendlyName()**

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

**current→get_functionDescriptor()**

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

**current→get_functionId()**

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

**current→get_hardwareId()**

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

**current→get_highestValue()**

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

**current→get_logFrequency()**

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

**current→get_logicalName()**

Returns the logical name of the current sensor.

**current→get_lowestValue()**

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

**current→get_module()**

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

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

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

**current→get_recordedData(startTime, endTime)**

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

**current→get_reportFrequency()**

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

**current→get_resolution()**

Returns the resolution of the measured values.

**current→get_unit()**

Returns the measuring unit for the current.

**current→get_userData()**

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

**current→isOnline()**

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

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

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

**current→load(msValidity)**

Preloads the current sensor cache with a specified validity duration.

**current→loadCalibrationPoints(rawValues, refValues)**

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

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

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

**current→nextCurrent()**

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

**current→registerTimedReportCallback(callback)**

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

**current→registerValueCallback(callback)**

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

**current→set_highestValue(newval)**

Changes the recorded maximal value observed.

**current→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**current→set_logicalName(newval)**

Changes the logical name of the current sensor.

**current→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**current→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**current→set_resolution(newval)**

Changes the resolution of the measured physical 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.

## 3.9. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

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

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

**Global functions**

**yFindDataLogger(func)**

Retrieves a data logger for a given identifier.

**yFirstDataLogger()**

Starts the enumeration of data loggers currently accessible.

**`YDataLogger` methods**

**datalogger→describe()**

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

**datalogger→forgetAllDataStreams()**

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

**datalogger→get_advertisedValue()**

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

**datalogger→get_autoStart()**

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

**datalogger→get_beaconDriven()**

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

**datalogger→get_currentRunIndex()**

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

**datalogger→get_dataSets()**

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

**datalogger→get_dataStreams(v)**

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

**datalogger→get_errorMessage()**

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

**datalogger→get_errorType()**

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

**datalogger→get_friendlyName()**

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_beaconDriven**(**newval**)

Changes the type of synchronisation of the data logger.

**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.

## 3.10. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

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

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

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

## datarun→get_startTimeUTC() datarun→startTimeUTC() YDataRun

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

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

**Returns :**

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

## 3.11. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instantiated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

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

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

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

| | |
|---|---|
| js | &lt;script type='text/javascript' src='yocto_api.js'&gt;&lt;/script&gt; |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>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 * |

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

**dataset→loadMore()**

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

**dataset→loadMore_async(callback, context)**

Loads the the next block of measures from the dataLogger asynchronously.

## 3.12. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

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

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

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

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

## 3.13. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

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

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

**Global functions**

**yFindDigitalIO(func)**

Retrieves a digital IO port for a given identifier.

**yFirstDigitalIO()**

Starts the enumeration of digital IO ports currently accessible.

**`YDigitalIO` methods**

**digitalio→delayedPulse(bitno, ms_delay, ms_duration)**

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

**digitalio→describe()**

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

**digitalio→get_advertisedValue()**

Returns the current value of the digital IO port (no more than 6 characters).

**digitalio→get_bitDirection(bitno)**

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

**digitalio→get_bitOpenDrain(bitno)**

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

**digitalio→get_bitPolarity(bitno)**

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

**digitalio→get_bitState(bitno)**

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

**digitalio→get_errorMessage()**

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

**digitalio→get_errorType()**

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

**digitalio→get_friendlyName()**

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

**digitalio→get_functionDescriptor()**

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

**digitalio→get_functionId()**

Returns the hardware identifier of the digital IO port, without reference to the module.

**digitalio→get_hardwareId()**

Returns the unique hardware identifier of the digital IO port in the form `SERIAL.FUNCTIONID`.

**digitalio→get_logicalName()**

Returns the logical name of the digital IO port.

**digitalio→get_module()**

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

**digitalio→get_module_async(callback, context)**

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

**digitalio→get_outputVoltage()**

Returns the voltage source used to drive output bits.

**digitalio→get_portDirection()**

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

**digitalio→get_portOpenDrain()**

Returns the electrical interface for each bit of the port.

**digitalio→get_portPolarity()**

Returns the polarity of all the bits of the port.

**digitalio→get_portSize()**

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

**digitalio→get_portState()**

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

**digitalio→get_userData()**

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

**digitalio→isOnline()**

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

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

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

**digitalio→load(msValidity)**

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

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

Preloads the digital IO port cache with a specified validity duration (asynchronous version).

**digitalio→nextDigitalIO()**

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

**digitalio→pulse(bitno, ms_duration)**

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

**digitalio→registerValueCallback(callback)**

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

**digitalio→set_bitDirection(bitno, bitdirection)**

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

**digitalio→set_bitOpenDrain(bitno, opendrain)**

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

**digitalio→set_bitPolarity(bitno, bitpolarity)**

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

**digitalio→set_bitState(bitno, bitstate)**

Sets a single bit of the I/O port.

**digitalio→set_logicalName(newval)**

Changes the logical name of the digital IO port.

**digitalio→set_outputVoltage(newval)**

Changes the voltage source used to drive output bits.

**digitalio→set_portDirection(newval)**

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

**digitalio→set_portOpenDrain(newval)**

Changes the electrical interface for each bit of the port.

**digitalio→set_portPolarity(newval)**

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

**digitalio→set_portState(newval)**

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

**digitalio→set_userData(data)**

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

**digitalio→toggle_bitState(bitno)**

Reverts a single bit of the I/O port.

**digitalio→wait_async(callback, context)**

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

## 3.14. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

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

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

**Global functions**

**yFindDisplay(func)**
Retrieves a display for a given identifier.

**yFirstDisplay()**
Starts the enumeration of displays currently accessible.

**`YDisplay` methods**

**display→copyLayerContent(srcLayerId, dstLayerId)**
Copies the whole content of a layer to another layer.

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

**display→fade(brightness, duration)**
Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

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

**display→get_brightness()**
Returns the luminosity of the module informative leds (from 0 to 100).

**display→get_displayHeight()**
Returns the display height, in pixels.

**display→get_displayLayer(layerId)**
Returns a YDisplayLayer object that can be used to draw on the specified layer.

**display→get_displayType()**
Returns the display type: monochrome, gray levels or full color.

**display→get_displayWidth()**
Returns the display width, in pixels.

**display→get_enabled()**
Returns true if the screen is powered, false otherwise.

**display→get_errorMessage()**
Returns the error message of the latest error with the display.

**display→get_errorType()**

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

**display→get_friendlyName()**

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

**display→get_functionDescriptor()**

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

**display→get_functionId()**

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

**display→get_hardwareId()**

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

**display→get_layerCount()**

Returns the number of available layers to draw on.

**display→get_layerHeight()**

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

**display→get_layerWidth()**

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

**display→get_logicalName()**

Returns the logical name of the display.

**display→get_module()**

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

**display→get_module_async(callback, context)**

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

**display→get_orientation()**

Returns the currently selected display orientation.

**display→get_startupSeq()**

Returns the name of the sequence to play when the displayed is powered on.

**display→get_userData()**

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

**display→isOnline()**

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

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

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

**display→load(msValidity)**

Preloads the display cache with a specified validity duration.

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

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

**display→newSequence()**

Starts to record all display commands into a sequence, for later replay.

**display→nextDisplay()**

Continues the enumeration of displays started using `yFirstDisplay()`.

**display→pauseSequence(delay_ms)**

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

**display→playSequence(sequenceName)**

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

**display→registerValueCallback(callback)**

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

**display→resetAll()**

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

**display→saveSequence(sequenceName)**

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

**display→set_brightness(newval)**

Changes the brightness of the display.

**display→set_enabled(newval)**

Changes the power state of the display.

**display→set_logicalName(newval)**

Changes the logical name of the display.

**display→set_orientation(newval)**

Changes the display orientation.

**display→set_startupSeq(newval)**

Changes the name of the sequence to play when the displayed is powered on.

**display→set_userData(data)**

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

**display→stopSequence()**

Stops immediately any ongoing sequence replay.

**display→swapLayerContent(layerIdA, layerIdB)**

Swaps the whole content of two layers.

**display→upload(pathname, content)**

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

**display→wait_async(callback, context)**

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

## 3.15. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_display.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YDisplay = yoctolib.YDisplay; |
| php | require_once('yocto_display.php'); |
| cpp | #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**()<br>Erases the whole content of the layer (makes it fully transparent). |
| **displaylayer→clearConsole**()<br>Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console. |
| **displaylayer→consoleOut**(**text**)<br>Outputs a message in the console area, and advances the console pointer accordingly. |
| **displaylayer→drawBar**(**x1**, **y1**, **x2**, **y2**)<br>Draws a filled rectangular bar at a specified position. |
| **displaylayer→drawBitmap**(**x**, **y**, **w**, **bitmap**, **bgcol**)<br>Draws a bitmap at the specified position. |
| **displaylayer→drawCircle**(**x**, **y**, **r**)<br>Draws an empty circle at a specified position. |
| **displaylayer→drawDisc**(**x**, **y**, **r**)<br>Draws a filled disc at a given position. |
| **displaylayer→drawImage**(**x**, **y**, **imagename**)<br>Draws a GIF image at the specified position. |
| **displaylayer→drawPixel**(**x**, **y**)<br>Draws a single pixel at the specified position. |
| **displaylayer→drawRect**(**x1**, **y1**, **x2**, **y2**)<br>Draws an empty rectangle at a specified position. |
| **displaylayer→drawText**(**x**, **y**, **anchor**, **text**)<br>Draws a text string at the specified position. |
| **displaylayer→get_display**()<br>Gets parent YDisplay. |
| **displaylayer→get_displayHeight**()<br>Returns the display height, in pixels. |
| **displaylayer→get_displayWidth**()<br>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 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.

## 3.16. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

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

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

**Global functions**

**yFindDualPower(func)**

Retrieves a dual power control for a given identifier.

**yFirstDualPower()**

Starts the enumeration of dual power controls currently accessible.

**`YDualPower` methods**

**dualpower→describe()**

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

**dualpower→get_advertisedValue()**

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

**dualpower→get_errorMessage()**

Returns the error message of the latest error with the power control.

**dualpower→get_errorType()**

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

**dualpower→get_extVoltage()**

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

**dualpower→get_friendlyName()**

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

**dualpower→get_functionDescriptor()**

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

**dualpower→get_functionId()**

Returns the hardware identifier of the power control, without reference to the module.

**dualpower→get_hardwareId()**

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

**dualpower→get_logicalName()**

Returns the logical name of the power control.

**dualpower→get_module()**

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

**dualpower→get_module_async(callback, context)**

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

**dualpower→get_powerControl()**

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

**dualpower→get_powerState()**

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

**dualpower→get_userData()**

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

**dualpower→isOnline()**

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

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

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

**dualpower→load(msValidity)**

Preloads the power control cache with a specified validity duration.

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

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

**dualpower→nextDualPower()**

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

**dualpower→registerValueCallback(callback)**

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

**dualpower→set_logicalName(newval)**

Changes the logical name of the power control.

**dualpower→set_powerControl(newval)**

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

**dualpower→set_userData(data)**

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

**dualpower→wait_async(callback, context)**

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

## 3.17. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

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

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

**Global functions**

**yFindFiles(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()**
Reinitialize the filesystem to its clean, unfragmented, empty state.

**files→get_advertisedValue()**
Returns the current value of the filesystem (no more than 6 characters).

**files→get_errorMessage()**
Returns the error message of the latest error with the filesystem.

**files→get_errorType()**
Returns the numerical error code of the latest error with the filesystem.

**files→get_filesCount()**
Returns the number of files currently loaded in the filesystem.

**files→get_freeSpace()**
Returns the free space for uploading new files to the filesystem, in bytes.

**files→get_friendlyName()**
Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

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

**files→get_functionId()**

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

**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.

## 3.18. GenericSensor function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_genericsensor.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YGenericSensor = yoctolib.YGenericSensor; |
| php | require_once('yocto_genericsensor.php'); |
| cpp | #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)**<br>Retrieves a generic sensor for a given identifier. |
| **yFirstGenericSensor()**<br>Starts the enumeration of generic sensors currently accessible. |
| **`YGenericSensor` methods** |
| **genericsensor→calibrateFromPoints(rawValues, refValues)**<br>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure. |
| **genericsensor→describe()**<br>Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **genericsensor→get_advertisedValue()**<br>Returns the current value of the generic sensor (no more than 6 characters). |
| **genericsensor→get_currentRawValue()**<br>Returns the uncalibrated, unrounded raw value returned by the sensor. |
| **genericsensor→get_currentValue()**<br>Returns the current measured value. |
| **genericsensor→get_errorMessage()**<br>Returns the error message of the latest error with the generic sensor. |
| **genericsensor→get_errorType()**<br>Returns the numerical error code of the latest error with the generic sensor. |
| **genericsensor→get_friendlyName()**<br>Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`. |
| **genericsensor→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **genericsensor→get_functionId()**<br>Returns the hardware identifier of the generic sensor, without reference to the module. |
| **genericsensor→get_hardwareId()**<br>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_signalBias()**
Returns the electric signal bias for zero shift adjustment.

**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_signalBias(newval)**

Changes the electric signal bias for zero shift adjustment.

**genericsensor→set_signalRange(newval)**

Changes the electric signal range used by the sensor.

**genericsensor→set_unit(newval)**

Changes the measuring unit for the measured value.

**genericsensor→set_userData(data)**

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

**genericsensor→set_valueRange(newval)**

Changes the physical value range measured by the sensor.

**genericsensor→wait_async(callback, context)**

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

**genericsensor→zeroAdjust()**

Adjusts the signal bias so that the current signal value is need precisely as zero.

## 3.19. Gyroscope function interface

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

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

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

**Global functions**

**yFindGyro(func)**
Retrieves a gyroscope for a given identifier.

**yFirstGyro()**
Starts the enumeration of gyroscopes currently accessible.

**`YGyro` methods**

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

**gyro→describe()**
Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

**gyro→get_advertisedValue()**
Returns the current value of the gyroscope (no more than 6 characters).

**gyro→get_currentRawValue()**
Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.

**gyro→get_currentValue()**
Returns the current value of the angular velocity, in degrees per second, as a floating point number.

**gyro→get_errorMessage()**
Returns the error message of the latest error with the gyroscope.

**gyro→get_errorType()**
Returns the numerical error code of the latest error with the gyroscope.

**gyro→get_friendlyName()**
Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.

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

**gyro→get_functionId()**
Returns the hardware identifier of the gyroscope, without reference to the module.

**gyro→get_hardwareId()**

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 `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_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()**

Returns the angular velocity around the Y axis of the device, as a floating point number.

**gyro→get_zValue()**

Returns the angular velocity around the Z axis of the device, as a floating point number.

**gyro→isOnline()**

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

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

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

**gyro→load(msValidity)**

Preloads the gyroscope cache with a specified validity duration.

**gyro→loadCalibrationPoints(rawValues, refValues)**

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

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

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

**gyro→nextGyro()**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

**gyro→registerAnglesCallback(callback)**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

**gyro→registerQuaternionCallback(callback)**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

**gyro→registerTimedReportCallback(callback)**

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

**gyro→registerValueCallback(callback)**

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

**gyro→set_highestValue(newval)**

Changes the recorded maximal value observed.

**gyro→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**gyro→set_logicalName(newval)**

Changes the logical name of the gyroscope.

**gyro→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**gyro→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**gyro→set_resolution(newval)**

Changes the resolution of the measured physical values.

**gyro→set_userData(data)**

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

**gyro→wait_async(callback, context)**

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

## 3.20. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

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

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

| Global functions |
|---|
| **yFindHubPort(func)**<br>Retrieves a Yocto-hub port for a given identifier. |
| **yFirstHubPort()**<br>Starts the enumeration of Yocto-hub ports currently accessible. |
| **`YHubPort` methods** |
| **hubport→describe()**<br>Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **hubport→get_advertisedValue()**<br>Returns the current value of the Yocto-hub port (no more than 6 characters). |
| **hubport→get_baudRate()**<br>Returns the current baud rate used by this Yocto-hub port, in kbps. |
| **hubport→get_enabled()**<br>Returns true if the Yocto-hub port is powered, false otherwise. |
| **hubport→get_errorMessage()**<br>Returns the error message of the latest error with the Yocto-hub port. |
| **hubport→get_errorType()**<br>Returns the numerical error code of the latest error with the Yocto-hub port. |
| **hubport→get_friendlyName()**<br>Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`. |
| **hubport→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **hubport→get_functionId()**<br>Returns the hardware identifier of the Yocto-hub port, without reference to the module. |
| **hubport→get_hardwareId()**<br>Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`. |
| **hubport→get_logicalName()**<br>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.

## 3.21. Humidity function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_humidity.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YHumidity = yoctolib.YHumidity; |
| php | require_once('yocto_humidity.php'); |
| cpp | #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)**<br>Retrieves a humidity sensor for a given identifier. |
| **yFirstHumidity()**<br>Starts the enumeration of humidity sensors currently accessible. |
| **`YHumidity` methods** |
| **humidity→calibrateFromPoints(rawValues, refValues)**<br>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure. |
| **humidity→describe()**<br>Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **humidity→get_advertisedValue()**<br>Returns the current value of the humidity sensor (no more than 6 characters). |
| **humidity→get_currentRawValue()**<br>Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number. |
| **humidity→get_currentValue()**<br>Returns the current value of the humidity, in %RH, as a floating point number. |
| **humidity→get_errorMessage()**<br>Returns the error message of the latest error with the humidity sensor. |
| **humidity→get_errorType()**<br>Returns the numerical error code of the latest error with the humidity sensor. |
| **humidity→get_friendlyName()**<br>Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`. |
| **humidity→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **humidity→get_functionId()**<br>Returns the hardware identifier of the humidity sensor, without reference to the module. |
| **humidity→get_hardwareId()**<br>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 since the device was started.

**humidity→get_logFrequency()**

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

**humidity→get_logicalName()**

Returns the logical name of the humidity sensor.

**humidity→get_lowestValue()**

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

**humidity→get_module()**

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

**humidity→get_module_async(callback, context)**

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

**humidity→get_recordedData(startTime, endTime)**

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

**humidity→get_reportFrequency()**

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

**humidity→get_resolution()**

Returns the resolution of the measured values.

**humidity→get_unit()**

Returns the measuring unit for the humidity.

**humidity→get_userData()**

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

**humidity→isOnline()**

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

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

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

**humidity→load(msValidity)**

Preloads the humidity sensor cache with a specified validity duration.

**humidity→loadCalibrationPoints(rawValues, refValues)**

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

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

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

**humidity→nextHumidity()**

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

**humidity→registerTimedReportCallback(callback)**

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

**humidity→registerValueCallback(callback)**

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

**humidity→set_highestValue(newval)**

Changes the recorded maximal value observed.

**humidity→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**humidity→set_logicalName(newval)**

Changes the logical name of the humidity sensor.

**humidity→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**humidity→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**humidity→set_resolution(newval)**

Changes the resolution of the measured physical values.

**humidity→set_userData(data)**

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

**humidity→wait_async(callback, context)**

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

## 3.22. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_led.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YLed = yoctolib.YLed; |
| php | require_once('yocto_led.php'); |
| cpp | #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)**<br>Retrieves a led for a given identifier. |
| **yFirstLed()**<br>Starts the enumeration of leds currently accessible. |
| **YLed methods** |
| **led→describe()**<br>Returns a short text that describes unambiguously the instance of the led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **led→get_advertisedValue()**<br>Returns the current value of the led (no more than 6 characters). |
| **led→get_blinking()**<br>Returns the current led signaling mode. |
| **led→get_errorMessage()**<br>Returns the error message of the latest error with the led. |
| **led→get_errorType()**<br>Returns the numerical error code of the latest error with the led. |
| **led→get_friendlyName()**<br>Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`. |
| **led→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **led→get_functionId()**<br>Returns the hardware identifier of the led, without reference to the module. |
| **led→get_hardwareId()**<br>Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`. |
| **led→get_logicalName()**<br>Returns the logical name of the led. |
| **led→get_luminosity()**<br>Returns the current led intensity (in per cent). |
| **led→get_module()** |

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

**led→get_module_async(callback, context)**

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

**led→get_power()**

Returns the current led state.

**led→get_userData()**

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

**led→isOnline()**

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

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

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

**led→load(msValidity)**

Preloads the led cache with a specified validity duration.

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

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

**led→nextLed()**

Continues the enumeration of leds started using `yFirstLed()`.

**led→registerValueCallback(callback)**

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

**led→set_blinking(newval)**

Changes the current led signaling mode.

**led→set_logicalName(newval)**

Changes the logical name of the led.

**led→set_luminosity(newval)**

Changes the current led intensity (in per cent).

**led→set_power(newval)**

Changes the state of the led.

**led→set_userData(data)**

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

**led→wait_async(callback, context)**

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

## 3.23. LightSensor function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_lightsensor.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YLightSensor = yoctolib.YLightSensor; |
| php | require_once('yocto_lightsensor.php'); |
| cpp | #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 uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.

**lightsensor→get_currentValue**()

Returns the current value of the ambient light, in lux, as a floating point number.

**lightsensor→get_errorMessage**()

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

**lightsensor→get_errorType**()

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

**lightsensor→get_friendlyName**()

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

**lightsensor→get_functionDescriptor**()

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

**lightsensor→get_functionId**()

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 since the device was started.

**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 since the device was started.

**lightsensor→get_measureType()**

Returns the type of light measure.

**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.

**lightsensor→set_logFrequency(newval)**

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.

**lightsensor→set_measureType(newval)**

Modify the light sensor type used in the device.

**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.

## 3.24. Magnetometer function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_magnetometer.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YMagnetometer = yoctolib.YMagnetometer; |
| php | require_once('yocto_magnetometer.php'); |
| cpp | #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, in mT, as a floating point number.

**magnetometer→get_currentValue()**

Returns the current value of the magnetic field, in mT, as a floating point number.

**magnetometer→get_errorMessage()**

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

**magnetometer→get_errorType()**

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

**magnetometer→get_friendlyName()**

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

**magnetometer→get_functionDescriptor()**

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

**magnetometer→get_functionId()**

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

**magnetometer→get_hardwareId()**

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

**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.

**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.

## 3.25. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>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()**<br>Returns the average value observed during the time interval covered by this measure. |
| **measure→get_endTimeUTC()**<br>Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp). |
| **measure→get_maxValue()**<br>Returns the largest value observed during the time interval covered by this measure. |
| **measure→get_minValue()**<br>Returns the smallest value observed during the time interval covered by this measure. |
| **measure→get_startTimeUTC()**<br>Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp). |

## 3.26. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>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→checkFirmware(path, onlynew)**
Test if the byn file is valid for this module.

**module→describe()**
Returns a descriptive text that identifies the module.

**module→download(pathname)**
Downloads the specified built-in file and returns a binary buffer with its content.

**module→functionCount()**
Returns the number of functions (beside the "module" interface) available on the module.

**module→functionId(functionIndex)**
Retrieves the hardware identifier of the *n*th function on the module.

**module→functionName(functionIndex)**
Retrieves the logical name of the *n*th function on the module.

**module→functionValue(functionIndex)**
Retrieves the advertised value of the *n*th function on the module.

**module→get_allSettings()**
Returns all the setting of the module.

**module→get_beacon()**
Returns the state of the localization beacon.

**module→get_errorMessage()**
Returns the error message of the latest error with this module object.

**module→get_errorType()**
Returns the numerical error code of the latest error with this module object.

**module→get_firmwareRelease()**

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_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→get_userVar**()

Returns the value previously stored in this attribute.

**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**)

Registers a device log callback function.

**module→revertFromFlash()**

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

**module→saveToFlash()**

Saves current settings in the nonvolatile memory of the module.

**module→set_allSettings(settings)**

Restore all the setting of the module.

**module→set_beacon(newval)**

Turns on or off the module localization beacon.

**module→set_logicalName(newval)**

Changes the logical name of the module.

**module→set_luminosity(newval)**

Changes the luminosity of the module informative leds.

**module→set_userData(data)**

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

**module→set_userVar(newval)**

Returns the value previously stored in this attribute.

**module→triggerFirmwareUpdate(secBeforeReboot)**

Schedules a module reboot into special firmware update mode.

**module→updateFirmware(path)**

Prepare a firmware upgrade of the module.

**module→wait_async(callback, context)**

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

## 3.27. Motor function interface

Yoctopuce application programming interface allows you to drive the power sent to the motor to make it turn both ways, but also to drive accelerations and decelerations. The motor will then accelerate automatically: you will not have to monitor it. The API also allows to slow down the motor by shortening its terminals: the motor will then act as an electromagnetic brake.

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

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

**Global functions**

**yFindMotor(func)**

Retrieves a motor for a given identifier.

**yFirstMotor()**

Starts the enumeration of motors currently accessible.

**`YMotor` methods**

**motor→brakingForceMove(targetPower, delay)**

Changes progressively the braking force applied to the motor for a specific duration.

**motor→describe()**

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

**motor→drivingForceMove(targetPower, delay)**

Changes progressively the power sent to the moteur for a specific duration.

**motor→get_advertisedValue()**

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

**motor→get_brakingForce()**

Returns the braking force applied to the motor, as a percentage.

**motor→get_cutOffVoltage()**

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

**motor→get_drivingForce()**

Returns the power sent to the motor, as a percentage between -100% and +100%.

**motor→get_errorMessage()**

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

**motor→get_errorType()**

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

**motor→get_failSafeTimeout()**

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

**motor→get_frequency()**
Returns the PWM frequency used to control the motor.

**motor→get_friendlyName()**
Returns a global identifier of the motor in the format `MODULE_NAME.FUNCTION_NAME`.

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

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

**motor→get_hardwareId()**
Returns the unique hardware identifier of the motor in the form `SERIAL.FUNCTIONID`.

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

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

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

**motor→get_motorStatus()**
Return the controller state.

**motor→get_overCurrentLimit()**
Returns the current threshold (in mA) above which the controller automatically switches to error state.

**motor→get_starterTime()**
Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

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

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

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

**motor→keepALive()**
Rearms the controller failsafe timer.

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

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

**motor→nextMotor()**
Continues the enumeration of motors started using `yFirstMotor()`.

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

**motor→resetStatus()**
Reset the controller state to IDLE.

**motor→set_brakingForce(newval)**
Changes immediately the braking force applied to the motor (in percents).

**motor→set_cutOffVoltage(newval)**
Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

**motor→set_drivingForce(newval)**

Changes immediately the power sent to the motor.

**motor→set_failSafeTimeout(newval)**

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

**motor→set_frequency(newval)**

Changes the PWM frequency used to control the motor.

**motor→set_logicalName(newval)**

Changes the logical name of the motor.

**motor→set_overCurrentLimit(newval)**

Changes the current threshold (in mA) above which the controller automatically switches to error state.

**motor→set_starterTime(newval)**

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

**motor→set_userData(data)**

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

**motor→wait_async(callback, context)**

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

## 3.28. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_network.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>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)**<br>Retrieves a network interface for a given identifier. |
| **yFirstNetwork()**<br>Starts the enumeration of network interfaces currently accessible. |
| **`YNetwork` methods** |
| **network→callbackLogin(username, password)**<br>Connects to the notification callback and saves the credentials required to log into it. |
| **network→describe()**<br>Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **network→get_adminPassword()**<br>Returns a hash string if a password has been set for user "admin", or an empty string otherwise. |
| **network→get_advertisedValue()**<br>Returns the current value of the network interface (no more than 6 characters). |
| **network→get_callbackCredentials()**<br>Returns a hashed version of the notification callback credentials if set, or an empty string otherwise. |
| **network→get_callbackEncoding()**<br>Returns the encoding standard to use for representing notification values. |
| **network→get_callbackMaxDelay()**<br>Returns the maximum waiting time between two callback notifications, in seconds. |
| **network→get_callbackMethod()**<br>Returns the HTTP method used to notify callbacks for significant state changes. |
| **network→get_callbackMinDelay()**<br>Returns the minimum waiting time between two callback notifications, in seconds. |
| **network→get_callbackUrl()**<br>Returns the callback URL to notify of significant state changes. |
| **network→get_discoverable()**<br>Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol). |

**network→get_errorMessage()**
Returns the error message of the latest error with the network interface.

**network→get_errorType()**
Returns the numerical error code of the latest error with the network interface.

**network→get_friendlyName()**
Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

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

**network→get_functionId()**
Returns the hardware identifier of the network interface, without reference to the module.

**network→get_hardwareId()**
Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

**network→get_ipAddress()**
Returns the IP address currently in use by the device.

**network→get_logicalName()**
Returns the logical name of the network interface.

**network→get_macAddress()**
Returns the MAC address of the network interface.

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

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

**network→get_poeCurrent()**
Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

**network→get_primaryDNS()**
Returns the IP address of the primary name server to be used by the module.

**network→get_readiness()**
Returns the current established working mode of the network interface.

**network→get_router()**
Returns the IP address of the router on the device subnet (default gateway).

**network→get_secondaryDNS()**
Returns the IP address of the secondary name server to be used by the module.

**network→get_subnetMask()**
Returns the subnet mask currently used by the device.

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

**network→get_userPassword()**
Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

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

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

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

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

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

**network→nextNetwork()**
Continues the enumeration of network interfaces started using `yFirstNetwork()`.

**network→ping(host)**
Pings str_host to test the network connectivity.

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

**network→set_adminPassword(newval)**
Changes the password for the "admin" user.

**network→set_callbackCredentials(newval)**
Changes the credentials required to connect to the callback address.

**network→set_callbackEncoding(newval)**
Changes the encoding standard to use for representing notification values.

**network→set_callbackMaxDelay(newval)**
Changes the maximum waiting time between two callback notifications, in seconds.

**network→set_callbackMethod(newval)**
Changes the HTTP method used to notify callbacks for significant state changes.

**network→set_callbackMinDelay(newval)**
Changes the minimum waiting time between two callback notifications, in seconds.

**network→set_callbackUrl(newval)**
Changes the callback URL to notify significant state changes.

**network→set_discoverable(newval)**
Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

**network→set_logicalName(newval)**
Changes the logical name of the network interface.

**network→set_primaryDNS(newval)**
Changes the IP address of the primary name server to be used by the module.

**network→set_secondaryDNS(newval)**
Changes the IP address of the secondary name server to be used by the module.

**network→set_userData(data)**
Stores a user context provided as argument in the userData attribute of the function.

**network→set_userPassword(newval)**
Changes the password for the "user" user.

**network→set_wwwWatchdogDelay(newval)**
Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

**network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)**
Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

**network→useStaticIP(ipAddress, subnetMaskLen, router)**
Changes the configuration of the network interface to use a static IP address.

**network→wait_async(callback, context)**

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

## 3.29. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

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

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

**Global functions**

**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.

## 3.30. Power function interface

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

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

| | |
|---|---|
| js | `<script type='text/javascript' src='yocto_power.js'></script>` |
| nodejs | var yoctolib = require('yoctolib');<br>var YPower = yoctolib.YPower; |
| php | require_once('yocto_power.php'); |
| cpp | #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, in Watt, as a floating point number.

**power→get_currentValue()**

Returns the current value of the electrical power, in Watt, as a floating point number.

**power→get_errorMessage()**

Returns the error message of the latest error with the electrical power sensor.

**power→get_errorType()**

Returns the numerical error code of the latest error with the electrical power sensor.

**power→get_friendlyName()**

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

**power→get_functionDescriptor()**

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

**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 since the device was started.

**power→get_logFrequency()**

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

**power→get_logicalName()**

Returns the logical name of the electrical power sensor.

**power→get_lowestValue()**

Returns the minimal value observed for the electrical power since the device was started.

**power→get_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

**power→get_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

**power→get_module()**

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

**power→get_module_async(callback, context)**

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

**power→get_recordedData(startTime, endTime)**

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

**power→get_reportFrequency()**

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

**power→get_resolution()**

Returns the resolution of the measured values.

**power→get_unit()**

Returns the measuring unit for the electrical power.

**power→get_userData()**

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

**power→isOnline()**

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

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

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

**power→load(msValidity)**

Preloads the electrical power sensor cache with a specified validity duration.

**power→loadCalibrationPoints(rawValues, refValues)**

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

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

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

**power→nextPower()**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

**power→registerTimedReportCallback(callback)**

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

**power→registerValueCallback(callback)**

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

**power→reset()**

Resets the energy counter.

**power→set_highestValue(newval)**

Changes the recorded maximal value observed.

**power→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**power→set_logicalName(newval)**

Changes the logical name of the electrical power sensor.

**power→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**power→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**power→set_resolution(newval)**

Changes the resolution of the measured physical values.

**power→set_userData(data)**

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

**power→wait_async(callback, context)**

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

## 3.31. Pressure function interface

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

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

| | |
|---|---|
| `js` | <script type='text/javascript' src='yocto_pressure.js'></script> |
| `nodejs` | var yoctolib = require('yoctolib');<br>var YPressure = yoctolib.YPressure; |
| `php` | require_once('yocto_pressure.php'); |
| `cpp` | #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 uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

**pressure→get_currentValue()**

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

**pressure→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 since the device was started.

**pressure→get_logFrequency**()

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

**pressure→get_logicalName**()

Returns the logical name of the pressure sensor.

**pressure→get_lowestValue**()

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

**pressure→get_module**()

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

**pressure→get_module_async**(**callback**, **context**)

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

**pressure→get_recordedData**(**startTime**, **endTime**)

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

**pressure→get_reportFrequency**()

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

**pressure→get_resolution**()

Returns the resolution of the measured values.

**pressure→get_unit**()

Returns the measuring unit for the pressure.

**pressure→get_userData**()

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

**pressure→isOnline**()

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

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

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

**pressure→load**(**msValidity**)

Preloads the pressure sensor cache with a specified validity duration.

**pressure→loadCalibrationPoints**(**rawValues**, **refValues**)

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

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

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

**pressure→nextPressure**()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

**pressure→registerTimedReportCallback**(**callback**)

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

**pressure→registerValueCallback**(**callback**)

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

**pressure→set_highestValue**(**newval**)

Changes the recorded maximal value observed.

**pressure→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**pressure→set_logicalName**(**newval**)

Changes the logical name of the pressure sensor.

**pressure→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**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.

## 3.32. PwmInput function interface

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

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

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

**Global functions**

**yFindPwmInput(func)**

Retrieves a voltage sensor for a given identifier.

**yFirstPwmInput()**

Starts the enumeration of voltage sensors currently accessible.

**`YPwmInput` methods**

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

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

**pwminput→describe()**

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

**pwminput→get_advertisedValue()**

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

**pwminput→get_currentRawValue()**

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

**pwminput→get_currentValue()**

Returns the current value of PwmInput feature as a floating point number.

**pwminput→get_dutyCycle()**

Returns the PWM duty cycle, in per cents.

**pwminput→get_errorMessage()**

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

**pwminput→get_errorType()**

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

**pwminput→get_frequency()**

Returns the PWM frequency in Hz.

**pwminput→get_friendlyName()**

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

**pwminput→get_functionDescriptor()**

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

**pwminput→get_functionId()**

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

**pwminput→get_hardwareId()**

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

**pwminput→get_highestValue()**

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

**pwminput→get_logFrequency()**

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

**pwminput→get_logicalName()**

Returns the logical name of the voltage sensor.

**pwminput→get_lowestValue()**

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

**pwminput→get_module()**

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

**pwminput→get_module_async(callback, context)**

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

**pwminput→get_period()**

Returns the PWM period in milliseconds.

**pwminput→get_pulseCounter()**

Returns the pulse counter value.

**pwminput→get_pulseDuration()**

Returns the PWM pulse length in milliseconds, as a floating point number.

**pwminput→get_pulseTimer()**

Returns the timer of the pulses counter (ms)

**pwminput→get_pwmReportMode()**

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.

**pwminput→get_recordedData(startTime, endTime)**

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

**pwminput→get_reportFrequency()**

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

**pwminput→get_resolution()**

Returns the resolution of the measured values.

**pwminput→get_unit()**

Returns the measuring unit for the values returned by get_currentValue and callbacks.

**pwminput→get_userData()**

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

**pwminput→isOnline()**

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

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

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

**pwminput→load(msValidity)**

Preloads the voltage sensor cache with a specified validity duration.

**pwminput→loadCalibrationPoints(rawValues, refValues)**

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

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

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

**pwminput→nextPwmInput()**

Continues the enumeration of voltage sensors started using `yFirstPwmInput()`.

**pwminput→registerTimedReportCallback(callback)**

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

**pwminput→registerValueCallback(callback)**

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

**pwminput→resetCounter()**

Returns the pulse counter value as well as his timer

**pwminput→set_highestValue(newval)**

Changes the recorded maximal value observed.

**pwminput→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**pwminput→set_logicalName(newval)**

Changes the logical name of the voltage sensor.

**pwminput→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**pwminput→set_pwmReportMode(newval)**

Modify the parameter type(frequency/duty cycle, pulse width ou edge count) returned by the get_currentValue function and callbacks.

**pwminput→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**pwminput→set_resolution(newval)**

Changes the resolution of the measured physical values.

**pwminput→set_userData(data)**

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

**pwminput→wait_async(callback, context)**

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

## 3.33. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

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

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

**Global functions**

**yFindPwmOutput(func)**

Retrieves a PWM for a given identifier.

**yFirstPwmOutput()**

Starts the enumeration of PWMs currently accessible.

**`YPwmOutput` methods**

**pwmoutput→describe()**

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

**pwmoutput→dutyCycleMove(target, ms_duration)**

Performs a smooth change of the pulse duration toward a given value.

**pwmoutput→get_advertisedValue()**

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

**pwmoutput→get_dutyCycle()**

Returns the PWM duty cycle, in per cents.

**pwmoutput→get_dutyCycleAtPowerOn()**

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

**pwmoutput→get_enabled()**

Returns the state of the PWMs.

**pwmoutput→get_enabledAtPowerOn()**

Returns the state of the PWM at device power on.

**pwmoutput→get_errorMessage()**

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

**pwmoutput→get_errorType()**

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

**pwmoutput→get_frequency()**

Returns the PWM frequency in Hz.

**pwmoutput→get_friendlyName()**

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

**pwmoutput→get_functionDescriptor()**

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

**pwmoutput→get_functionId()**

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

**pwmoutput→get_hardwareId()**

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

**pwmoutput→get_logicalName()**

Returns the logical name of the PWM.

**pwmoutput→get_module()**

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

**pwmoutput→get_module_async(callback, context)**

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

**pwmoutput→get_period()**

Returns the PWM period in milliseconds.

**pwmoutput→get_pulseDuration()**

Returns the PWM pulse length in milliseconds, as a floating point number.

**pwmoutput→get_userData()**

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

**pwmoutput→isOnline()**

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

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

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

**pwmoutput→load(msValidity)**

Preloads the PWM cache with a specified validity duration.

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

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

**pwmoutput→nextPwmOutput()**

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

**pwmoutput→pulseDurationMove(ms_target, ms_duration)**

Performs a smooth transistion of the pulse duration toward a given value.

**pwmoutput→registerValueCallback(callback)**

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

**pwmoutput→set_dutyCycle(newval)**

Changes the PWM duty cycle, in per cents.

**pwmoutput→set_dutyCycleAtPowerOn(newval)**

Changes the PWM duty cycle at device power on.

**pwmoutput→set_enabled(newval)**

Stops or starts the PWM.

**pwmoutput→set_enabledAtPowerOn(newval)**

Changes the state of the PWM at device power on.

**pwmoutput→set_frequency(newval)**

Changes the PWM frequency.

**pwmoutput→set_logicalName(newval)**

Changes the logical name of the PWM.

**pwmoutput→set_period(newval)**

Changes the PWM period in milliseconds.

**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.

## 3.34. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

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

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

| Global functions |
|---|
| **yFindPwmPowerSource(func)**<br>Retrieves a voltage source for a given identifier. |
| **yFirstPwmPowerSource()**<br>Starts the enumeration of Voltage sources currently accessible. |
| **`YPwmPowerSource` methods** |
| **pwmpowersource→describe()**<br>Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **pwmpowersource→get_advertisedValue()**<br>Returns the current value of the voltage source (no more than 6 characters). |
| **pwmpowersource→get_errorMessage()**<br>Returns the error message of the latest error with the voltage source. |
| **pwmpowersource→get_errorType()**<br>Returns the numerical error code of the latest error with the voltage source. |
| **pwmpowersource→get_friendlyName()**<br>Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`. |
| **pwmpowersource→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **pwmpowersource→get_functionId()**<br>Returns the hardware identifier of the voltage source, without reference to the module. |
| **pwmpowersource→get_hardwareId()**<br>Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`. |
| **pwmpowersource→get_logicalName()**<br>Returns the logical name of the voltage source. |
| **pwmpowersource→get_module()**<br>Gets the `YModule` object for the device on which the function is located. |
| **pwmpowersource→get_module_async(callback, context)**<br>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.

## 3.35. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

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

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

**Global functions**

**yFindQt(func)**
Retrieves a quaternion component for a given identifier.

**yFirstQt()**
Starts the enumeration of quaternion components currently accessible.

**YQt methods**

**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, in units, as a floating point number.

**qt→get_currentValue()**
Returns the current value of the value, in units, as a floating point number.

**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)**

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.

## 3.36. 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');<br>var YRealTimeClock = yoctolib.YRealTimeClock; |
| php | require_once('yocto_realtimeclock.php'); |
| cpp | #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 `YModule` object for the device on which the function is located.

**realtimeclock→get_module_async(callback, context)**

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

**realtimeclock→get_timeSet()**

Returns true if the clock has been set, and false otherwise.

**realtimeclock→get_unixTime()**

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

**realtimeclock→get_userData()**

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

**realtimeclock→get_utcOffset()**

Returns the number of seconds between current time and UTC time (time zone).

**realtimeclock→isOnline()**

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

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

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

**realtimeclock→load(msValidity)**

Preloads the clock cache with a specified validity duration.

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

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

**realtimeclock→nextRealTimeClock()**

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

**realtimeclock→registerValueCallback(callback)**

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

**realtimeclock→set_logicalName(newval)**

Changes the logical name of the clock.

**realtimeclock→set_unixTime(newval)**

Changes the current time.

**realtimeclock→set_userData(data)**

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

**realtimeclock→set_utcOffset(newval)**

Changes the number of seconds between current time and UTC time (time zone).

**realtimeclock→wait_async(callback, context)**

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

## 3.37. 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');<br>var YRefFrame = yoctolib.YRefFrame; |
| php | require_once('yocto_refframe.php'); |
| cpp | #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)**

Changes the logical name of the reference frame.

**refframe→set_mountPosition(position, orientation)**

Changes the compass and tilt sensor frame of reference.

**refframe→set_userData(data)**

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

**refframe→start3DCalibration()**

Initiates the sensors tridimensional calibration process.

**refframe→wait_async(callback, context)**

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

## 3.38. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

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

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

| Global functions |
|---|
| **yFindRelay(func)**<br>Retrieves a relay for a given identifier. |
| **yFirstRelay()**<br>Starts the enumeration of relays currently accessible. |
| **`YRelay` methods** |
| **relay→delayedPulse(ms_delay, ms_duration)**<br>Schedules a pulse. |
| **relay→describe()**<br>Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **relay→get_advertisedValue()**<br>Returns the current value of the relay (no more than 6 characters). |
| **relay→get_countdown()**<br>Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero. |
| **relay→get_errorMessage()**<br>Returns the error message of the latest error with the relay. |
| **relay→get_errorType()**<br>Returns the numerical error code of the latest error with the relay. |
| **relay→get_friendlyName()**<br>Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`. |
| **relay→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **relay→get_functionId()**<br>Returns the hardware identifier of the relay, without reference to the module. |
| **relay→get_hardwareId()** |

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

**relay→get_logicalName()**

Returns the logical name of the relay.

**relay→get_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

**relay→get_maxTimeOnStateB()**

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

**relay→get_module()**

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

**relay→get_module_async(callback, context)**

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

**relay→get_output()**

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

**relay→get_pulseTimer()**

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

**relay→get_state()**

Returns the state of the relays (A for the idle position, B for the active position).

**relay→get_stateAtPowerOn()**

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

**relay→get_userData()**

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

**relay→isOnline()**

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

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

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

**relay→load(msValidity)**

Preloads the relay cache with a specified validity duration.

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

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

**relay→nextRelay()**

Continues the enumeration of relays started using `yFirstRelay()`.

**relay→pulse(ms_duration)**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

**relay→registerValueCallback(callback)**

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

**relay→set_logicalName(newval)**

Changes the logical name of the relay.

**relay→set_maxTimeOnStateA(newval)**

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

**relay→set_maxTimeOnStateB(newval)**

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

**relay→set_output(newval)**

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

**relay→set_state(newval)**

Changes the state of the relays (A for the idle position, B for the active position).

**relay→set_stateAtPowerOn(newval)**

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

**relay→set_userData(data)**

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

**relay→wait_async(callback, context)**

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

## 3.39. Sensor function interface

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

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

| | |
|---|---|
| `js` | <script type='text/javascript' src='yocto_api.js'></script> |
| `nodejs` | var yoctolib = require('yoctolib');<br>var YAPI = yoctolib.YAPI;<br>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)**<br>Retrieves a sensor for a given identifier. |
| **yFirstSensor()**<br>Starts the enumeration of sensors currently accessible. |
| **`YSensor` methods** |
| **sensor→calibrateFromPoints(rawValues, refValues)**<br>Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure. |
| **sensor→describe()**<br>Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **sensor→get_advertisedValue()**<br>Returns the current value of the sensor (no more than 6 characters). |
| **sensor→get_currentRawValue()**<br>Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number. |
| **sensor→get_currentValue()**<br>Returns the current value of the measure, in the specified unit, as a floating point number. |
| **sensor→get_errorMessage()**<br>Returns the error message of the latest error with the sensor. |
| **sensor→get_errorType()**<br>Returns the numerical error code of the latest error with the sensor. |
| **sensor→get_friendlyName()**<br>Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`. |
| **sensor→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **sensor→get_functionId()**<br>Returns the hardware identifier of the sensor, without reference to the module. |
| **sensor→get_hardwareId()** |

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

**sensor→get_highestValue()**

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

**sensor→get_logFrequency()**

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

**sensor→get_logicalName()**

Returns the logical name of the sensor.

**sensor→get_lowestValue()**

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

**sensor→get_module()**

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

**sensor→get_module_async(callback, context)**

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

**sensor→get_recordedData(startTime, endTime)**

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

**sensor→get_reportFrequency()**

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

**sensor→get_resolution()**

Returns the resolution of the measured values.

**sensor→get_unit()**

Returns the measuring unit for the measure.

**sensor→get_userData()**

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

**sensor→isOnline()**

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

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

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

**sensor→load(msValidity)**

Preloads the sensor cache with a specified validity duration.

**sensor→loadCalibrationPoints(rawValues, refValues)**

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

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

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

**sensor→nextSensor()**

Continues the enumeration of sensors started using `yFirstSensor()`.

**sensor→registerTimedReportCallback(callback)**

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

**sensor→registerValueCallback(callback)**

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

**sensor→set_highestValue(newval)**

Changes the recorded maximal value observed.

**sensor→set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**sensor→set_logicalName(newval)**

Changes the logical name of the sensor.

**sensor→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**sensor→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**sensor→set_resolution(newval)**

Changes the resolution of the measured physical values.

**sensor→set_userData(data)**

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

**sensor→wait_async(callback, context)**

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

## 3.40. SerialPort function interface

The SerialPort function interface allows you to fully drive a Yoctopuce serial port, to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

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

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

**Global functions**

**yFindSerialPort(func)**

Retrieves a serial port for a given identifier.

**yFirstSerialPort()**

Starts the enumeration of serial ports currently accessible.

**`YSerialPort` methods**

**serialport→describe()**

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

**serialport→get_CTS()**

Read the level of the CTS line.

**serialport→get_advertisedValue()**

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

**serialport→get_errCount()**

Returns the total number of communication errors detected since last reset.

**serialport→get_errorMessage()**

Returns the error message of the latest error with the serial port.

**serialport→get_errorType()**

Returns the numerical error code of the latest error with the serial port.

**serialport→get_friendlyName()**

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

**serialport→get_functionDescriptor()**

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

**serialport→get_functionId()**

Returns the hardware identifier of the serial port, without reference to the module.

**serialport→get_hardwareId()**

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

**serialport→get_lastMsg()**

Returns the latest message fully received (for Line, Frame and Modbus protocols).

**serialport→get_logicalName()**

Returns the logical name of the serial port.

**serialport→get_module()**

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

**serialport→get_module_async(callback, context)**

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

**serialport→get_msgCount()**

Returns the total number of messages received since last reset.

**serialport→get_protocol()**

Returns the type of protocol used over the serial line, as a string.

**serialport→get_rxCount()**

Returns the total number of bytes received since last reset.

**serialport→get_serialMode()**

Returns the serial port communication parameters, as a string such as "9600,8N1".

**serialport→get_txCount()**

Returns the total number of bytes transmitted since last reset.

**serialport→get_userData()**

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

**serialport→isOnline()**

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

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

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

**serialport→load(msValidity)**

Preloads the serial port cache with a specified validity duration.

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

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

**serialport→modbusReadBits(slaveNo, pduAddr, nBits)**

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

**serialport→modbusReadInputBits(slaveNo, pduAddr, nBits)**

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

**serialport→modbusReadInputRegisters(slaveNo, pduAddr, nWords)**

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

**serialport→modbusReadRegisters(slaveNo, pduAddr, nWords)**

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

**serialport→modbusWriteAndReadRegisters(slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)**

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

**serialport→modbusWriteBit(slaveNo, pduAddr, value)**

Sets a single internal bit (or coil) on a MODBUS serial device.

**serialport→modbusWriteBits(slaveNo, pduAddr, bits)**

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

**serialport→modbusWriteRegister(slaveNo, pduAddr, value)**

Sets a single internal register (or holding register) on a MODBUS serial device.

**serialport→modbusWriteRegisters(slaveNo, pduAddr, values)**

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

**serialport→nextSerialPort()**

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

**serialport→queryLine(query, maxWait)**

Sends a text line query to the serial port, and reads the reply, if any.

**serialport→queryMODBUS(slaveNo, pduBytes)**

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

**serialport→readHex(nBytes)**

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

**serialport→readLine()**

Reads a single line (or message) from the receive buffer, starting at current stream position.

**serialport→readMessages(pattern, maxWait)**

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

**serialport→readStr(nChars)**

Reads data from the receive buffer as a string, starting at current stream position.

**serialport→read_seek(rxCountVal)**

Changes the current internal stream position to the specified value.

**serialport→registerValueCallback(callback)**

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

**serialport→reset()**

Clears the serial port buffer and resets counters to zero.

**serialport→set_RTS(val)**

Manually sets the state of the RTS line.

**serialport→set_logicalName(newval)**

Changes the logical name of the serial port.

**serialport→set_protocol(newval)**

Changes the type of protocol used over the serial line.

**serialport→set_serialMode(newval)**

Changes the serial port communication parameters, with a string such as "9600,8N1".

**serialport→set_userData(data)**

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

**serialport→wait_async(callback, context)**

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

**serialport→writeArray(byteList)**

Sends a byte sequence (provided as a list of bytes) to the serial port.

**serialport→writeBin(buff)**

Sends a binary buffer to the serial port, as is.

**serialport→writeHex(hexString)**

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

**serialport→writeLine(text)**

Sends an ASCII string to the serial port, followed by a line break (CR LF).

**serialport→writeMODBUS(hexString)**

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

**serialport→writeStr(text)**

Sends an ASCII string to the serial port, as is.

## 3.41. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_servo.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YServo = yoctolib.YServo; |
| php | require_once('yocto_servo.php'); |
| cpp | #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**)<br>Retrieves a servo for a given identifier. |
| **yFirstServo**()<br>Starts the enumeration of servos currently accessible. |
| **`YServo` methods** |
| **servo→describe**()<br>Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **servo→get_advertisedValue**()<br>Returns the current value of the servo (no more than 6 characters). |
| **servo→get_enabled**()<br>Returns the state of the servos. |
| **servo→get_enabledAtPowerOn**()<br>Returns the servo signal generator state at power up. |
| **servo→get_errorMessage**()<br>Returns the error message of the latest error with the servo. |
| **servo→get_errorType**()<br>Returns the numerical error code of the latest error with the servo. |
| **servo→get_friendlyName**()<br>Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`. |
| **servo→get_functionDescriptor**()<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **servo→get_functionId**()<br>Returns the hardware identifier of the servo, without reference to the module. |
| **servo→get_hardwareId**()<br>Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`. |
| **servo→get_logicalName**()<br>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.

## 3.42. Temperature function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_temperature.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YTemperature = yoctolib.YTemperature; |
| php | require_once('yocto_temperature.php'); |
| cpp | #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, in Celsius, as a floating point number.

**temperature→get_currentValue()**

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

**temperature→get_errorMessage()**

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

**temperature→get_errorType()**

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

**temperature→get_friendlyName()**

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

**temperature→get_functionDescriptor()**

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

**temperature→get_functionId()**

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

**temperature→get_hardwareId()**

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

**temperature→get_highestValue**()

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

**temperature→get_logFrequency**()

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

**temperature→get_logicalName**()

Returns the logical name of the temperature sensor.

**temperature→get_lowestValue**()

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

**temperature→get_module**()

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

**temperature→get_module_async**(**callback**, **context**)

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

**temperature→get_recordedData**(**startTime**, **endTime**)

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

**temperature→get_reportFrequency**()

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

**temperature→get_resolution**()

Returns the resolution of the measured values.

**temperature→get_sensorType**()

Returns the temperature sensor type.

**temperature→get_unit**()

Returns the measuring unit for the temperature.

**temperature→get_userData**()

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

**temperature→isOnline**()

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

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

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

**temperature→load**(**msValidity**)

Preloads the temperature sensor cache with a specified validity duration.

**temperature→loadCalibrationPoints**(**rawValues**, **refValues**)

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

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

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

**temperature→nextTemperature**()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

**temperature→registerTimedReportCallback**(**callback**)

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

**temperature→registerValueCallback**(**callback**)

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

**temperature→set_highestValue**(**newval**)

Changes the recorded maximal value observed.

**temperature→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**temperature→set_logicalName(newval)**

Changes the logical name of the temperature sensor.

**temperature→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**temperature→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**temperature→set_resolution(newval)**

Changes the resolution of the measured physical values.

**temperature→set_sensorType(newval)**

Modify the temperature sensor type.

**temperature→set_userData(data)**

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

**temperature→wait_async(callback, context)**

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

## 3.43. Tilt function interface

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

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

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

**Global functions**

**yFindTilt(func)**
Retrieves a tilt sensor for a given identifier.

**yFirstTilt()**
Starts the enumeration of tilt sensors currently accessible.

**YTilt methods**

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

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

**tilt→get_advertisedValue()**
Returns the current value of the tilt sensor (no more than 6 characters).

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

**tilt→get_currentValue()**
Returns the current value of the inclination, in degrees, as a floating point number.

**tilt→get_errorMessage()**
Returns the error message of the latest error with the tilt sensor.

**tilt→get_errorType()**
Returns the numerical error code of the latest error with the tilt sensor.

**tilt→get_friendlyName()**
Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

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

**tilt→get_functionId()**
Returns the hardware identifier of the tilt sensor, without reference to the module.

**tilt→get_hardwareId()**
Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

**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.

## 3.44. Voc function interface

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

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

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

**Global functions**

**yFindVoc(func)**

Retrieves a Volatile Organic Compound sensor for a given identifier.

**yFirstVoc()**

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

**YVoc methods**

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

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

**voc→describe()**

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

**voc→get_advertisedValue()**

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

**voc→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

**voc→get_currentValue()**

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

**voc→get_errorMessage()**

Returns the error message of the latest error with the Volatile Organic Compound sensor.

**voc→get_errorType()**

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

**voc→get_friendlyName()**

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

**voc→get_functionDescriptor()**

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

**voc→get_functionId()**

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

**voc→get_hardwareId()**

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

**voc→get_highestValue()**

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

**voc→get_logFrequency()**

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

**voc→get_logicalName()**

Returns the logical name of the Volatile Organic Compound sensor.

**voc→get_lowestValue()**

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

**voc→get_module()**

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

**voc→get_module_async(callback, context)**

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

**voc→get_recordedData(startTime, endTime)**

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

**voc→get_reportFrequency()**

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

**voc→get_resolution()**

Returns the resolution of the measured values.

**voc→get_unit()**

Returns the measuring unit for the estimated VOC concentration.

**voc→get_userData()**

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

**voc→isOnline()**

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

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

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

**voc→load(msValidity)**

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

**voc→loadCalibrationPoints(rawValues, refValues)**

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

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

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

**voc→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.

**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.

**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.

## 3.45. Voltage function interface

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

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_voltage.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YVoltage = yoctolib.YVoltage; |
| php | require_once('yocto_voltage.php'); |
| cpp | #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, in Volt, as a floating point number.

**voltage→get_currentValue()**

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

**voltage→get_errorMessage()**

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

**voltage→get_errorType()**

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

**voltage→get_friendlyName()**

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

**voltage→get_functionDescriptor()**

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

**voltage→get_functionId()**

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

**voltage→get_hardwareId()**

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

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

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

**voltage→get_logicalName()**
Returns the logical name of the voltage sensor.

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

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

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

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

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

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

**voltage→get_unit()**
Returns the measuring unit for the voltage.

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

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

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

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

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

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

**voltage→nextVoltage()**
Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

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

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

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

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

**voltage→set_logicalName(newval)**
Changes the logical name of the voltage sensor.

**voltage→set_lowestValue(newval)**

Changes the recorded minimal value observed.

**voltage→set_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**voltage→set_resolution(newval)**

Changes the resolution of the measured physical 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.

## 3.46. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

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

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

**Global functions**

**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 `TYPE(NAME)=SERIAL.FUNCTIONID`.

**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 `MODULE_NAME.FUNCTION_NAME`.

**vsource→get_functionDescriptor()**
Returns a unique identifier of type `YFUN_DESCR` 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 `SERIAL.FUNCTIONID`.

**vsource→get_logicalName()**
Returns the logical name of the voltage source.

**vsource→get_module()**
Gets the `YModule` 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).

**vsource→get_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

**vsource→get_overHeat()**

Returns TRUE if the module is overheating.

**vsource→get_overLoad()**

Returns true if the device is not able to maintaint the requested voltage output .

**vsource→get_regulationFailure()**

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

**vsource→get_unit()**

Returns the measuring unit for the voltage.

**vsource→get_userData()**

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

**vsource→get_voltage()**

Returns the voltage output command (mV)

**vsource→isOnline()**

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

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

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

**vsource→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

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

**vsource→nextVSource()**

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

**vsource→pulse(voltage, ms_duration)**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

**vsource→registerValueCallback(callback)**

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

**vsource→set_logicalName(newval)**

Changes the logical name of the voltage source.

**vsource→set_userData(data)**

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

**vsource→set_voltage(newval)**

Tunes the device output voltage (milliVolts).

**vsource→voltageMove(target, ms_duration)**

Performs a smooth move at constant speed toward a given value.

**vsource→wait_async(callback, context)**

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

## 3.47. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_wakeupmonitor.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YWakeUpMonitor = yoctolib.YWakeUpMonitor; |
| php | require_once('yocto_wakeupmonitor.php'); |
| cpp | #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**)

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

**wakeupmonitor→wakeUp()**

Forces a wake up.

## 3.48. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

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

| | |
|---|---|
| `js` | <script type='text/javascript' src='yocto_wakeupschedule.js'></script> |
| `nodejs` | var yoctolib = require('yoctolib');<br>var YWakeUpSchedule = yoctolib.YWakeUpSchedule; |
| `php` | require_once('yocto_wakeupschedule.php'); |
| `cpp` | #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)**<br>Retrieves a wake up schedule for a given identifier. |
| **yFirstWakeUpSchedule()**<br>Starts the enumeration of wake up schedules currently accessible. |
| **`YWakeUpSchedule` methods** |
| **wakeupschedule→describe()**<br>Returns a short text that describes unambiguously the instance of the wake up schedule in the form `TYPE(NAME)=SERIAL.FUNCTIONID`. |
| **wakeupschedule→get_advertisedValue()**<br>Returns the current value of the wake up schedule (no more than 6 characters). |
| **wakeupschedule→get_errorMessage()**<br>Returns the error message of the latest error with the wake up schedule. |
| **wakeupschedule→get_errorType()**<br>Returns the numerical error code of the latest error with the wake up schedule. |
| **wakeupschedule→get_friendlyName()**<br>Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`. |
| **wakeupschedule→get_functionDescriptor()**<br>Returns a unique identifier of type `YFUN_DESCR` corresponding to the function. |
| **wakeupschedule→get_functionId()**<br>Returns the hardware identifier of the wake up schedule, without reference to the module. |
| **wakeupschedule→get_hardwareId()**<br>Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`. |
| **wakeupschedule→get_hours()**<br>Returns the hours scheduled for wake up. |
| **wakeupschedule→get_logicalName()**<br>Returns the logical name of the wake up schedule. |
| **wakeupschedule→get_minutes()**<br>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 intervalof 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 occurence

**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.

## 3.49. Watchdog function interface

The watchog 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 direcly 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');<br>var YWatchdog = yoctolib.YWatchdog; |
| php | require_once('yocto_watchdog.php'); |
| cpp | #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 runing 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 runningsttae 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.

## 3.50. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

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

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_wireless.js'></script> |
| nodejs | var yoctolib = require('yoctolib');<br>var YWireless = yoctolib.YWireless; |
| php | require_once('yocto_wireless.php'); |
| cpp | #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→softAPNetwork(ssid, securityKey)**

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

**wireless→wait_async(callback, context)**

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

# Index