



Delphi API Reference

Table of contents

1. Introduction	1
2. Using Yocto-Demo with Delphi	3
2.1. Preparation	3
2.2. Control of the Led function	3
2.3. Control of the module part	5
2.4. Error handling	7
Blueprint	10
3. Reference	10
3.1. General functions	11
3.2. Accelerometer function interface	31
3.3. AnButton function interface	70
3.4. CarbonDioxide function interface	105
3.5. ColorLed function interface	141
3.6. Compass function interface	167
3.7. Current function interface	204
3.8. DataLogger function interface	240
3.9. Formatted data sequence	268
3.10. Recorded data sequence	270
3.11. Unformatted data sequence	282
3.12. Digital IO function interface	297
3.13. Display function interface	338
3.14. DisplayLayer object interface	381
3.15. External power supply control interface	413
3.16. Files function interface	435
3.17. GenericSensor function interface	460
3.18. Gyroscope function interface	503
3.19. Yocto-hub port interface	551
3.20. Humidity function interface	573
3.21. Led function interface	609
3.22. LightSensor function interface	633
3.23. Magnetometer function interface	670
3.24. Measured value	709
3.25. Module control interface	715

3.26. Network function interface	756
3.27. OS control	810
3.28. Power function interface	830
3.29. Pressure function interface	870
3.30. Pwm function interface	906
3.31. PwmPowerSource function interface	941
3.32. Quaternion interface	961
3.33. Real Time Clock function interface	997
3.34. Reference frame configuration	1021
3.35. Relay function interface	1054
3.36. Sensor function interface	1087
3.37. Servo function interface	1123
3.38. Temperature function interface	1155
3.39. Tilt function interface	1193
3.40. Voc function interface	1229
3.41. Voltage function interface	1265
3.42. Voltage source function interface	1301
3.43. WakeUpMonitor function interface	1330
3.44. WakeUpSchedule function interface	1362
3.45. Watchdog function interface	1396
3.46. Wireless function interface	1438

Index	1465
--------------------	-------------

1. Introduction

This manual is intended to be used as a reference for Yoctopuce Delphi 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 Delphi

Delphi is a descendent of Turbo-Pascal. Originally, Delphi was produced by Borland, Embarcadero now edits it. The strength of this language resides in its ease of use, as anyone with some notions of the Pascal language can develop a Windows application in next to no time. Its only disadvantage is to cost something¹.

Delphi libraries are provided not as VCL components, but directly as source files. These files are compatible with most Delphi versions.²

To keep them simple, all the examples provided in this documentation are console applications. Obviously, the libraries work in a strictly identical way with VCL applications.

You will soon notice that the Delphi 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.

2.1. Preparation

Go to the Yoctopuce web site and download the Yoctopuce Delphi libraries³. Uncompress everything in a directory of your choice, add the subdirectory *sources* in the list of directories of Delphi libraries.⁴

By default, the Yoctopuce Delphi library uses the *yapi.dll* DLL, all the applications you will create with Delphi must have access to this DLL. The simplest way to ensure this is to make sure *yapi.dll* is located in the same directory as the executable file of your application.

2.2. Control of the Led function

Launch your Delphi environment, copy the *yapi.dll* DLL in a directory, create a new console application in the same directory, and copy-paste the piece of code below:

```
program helloworld;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  yocto_api,
  yocto_led;
```

¹ Actually, Borland provided free versions (for personal use) of Delphi 2006 and 2007. Look for them on the Internet, you may still be able to download them.

² Delphi libraries are regularly tested with Delphi 5 and Delphi XE2.

³ www.yoctopuce.com/EN/libraries.php

⁴ Use the **Tools / Environment options** menu.

```

Procedure Usage();
var
  exe : string;

begin
  exe:= ExtractFileName(paramstr(0));
  WriteLn(exe+' <serial_number>');
  WriteLn(exe+' <logical_name>');
  WriteLn(exe+' any');
  halt;
End;

procedure setLedState(led:TYLed; state:boolean);
begin
  if (led.isOnline()) then
  begin
    if state then led.set_power(Y_POWER_ON)
    else led.set_power(Y_POWER_OFF);
  end
  else Writeln('Module not connected (check identification and USB cable)');
end;

var
  c      : char;
  led    : TYLed;
  errmsg : string;

begin
  // Setup the API to use local USB devices
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
    Write('RegisterHub error: '+errmsg);
    exit;
  end;

  if paramstr(1)='any' then
  begin
    // use the first available led
    led := yFirstLed();
    if led=nil then
    begin
      writeln('No module connected (check USB cable)');
      halt;
    end
  end
  else // or use the one specified on the command line
    led:= YFindLed(paramstr(1)+'.led');

  // make sure it is connected
  if not(led.isOnline()) then
  begin
    Writeln('Module not connected (check identification and USB cable)');
    halt;
  end;

  // minimalist UI
  Writeln('0: turn test led OFF');
  Writeln('1: turn test led ON');
  Writeln('x: exit');
  repeat
    read(c);
    case c of
      '0' : setLedState(led,false);
      '1' : setLedState(led,true);
    end;
  until c='x';

end.

```

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

yocto_api and yocto_led

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

yRegisterHub

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

yFindLed

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

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

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

isOnline

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

set_power

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

2.3. 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.

```
program modulecontrol;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  yocto_api;

const
  serial = 'YCTOPOC1-123456'; // use serial number or logical name

procedure refresh(module:Tymodule) ;
begin
  if (module.isOnline()) then
    begin
      Writeln('');
      Writeln('Serial      : ' + module.get_serialNumber());
      Writeln('Logical name : ' + module.get_logicalName());
      Writeln('Luminosity  : ' + intToStr(module.get_luminosity()));
      Write('Beacon    :');
      if (module.get_beacon()=Y_BEACON_ON) then Writeln('on')
      else Writeln('off');
      Writeln('uptime     : ' + intToStr(module.get_upTime() div 1000)+'s');
      Writeln('USB current : ' + intToStr(module.get_usbCurrent())+'mA');
      Writeln('Logs       : ');
      Writeln(module.get_lastlogs());
      Writeln('');
      Writeln('r : refresh / b:beacon ON / space : beacon off');
    end
  else Writeln('Module not connected (check identification and USB cable)');
end;
```

```

procedure beacon(module:Tymodule;state:integer);
begin
  module.set_beacon(state);
  refresh(module);
end;

var
  module : TYModule;
  c       : char;
  errmsg  : string;

begin
  // Setup the API to use local USB devices
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
    Write('RegisterHub error: '+errmsg);
    exit;
  end;

  module := yFindModule(serial);
  refresh(module);

  repeat
    read(c);
    case c of
      'r': refresh(module);
      'b': beacon(module,Y_BEACON_ON);
      ' ': beacon(module,Y_BEACON_OFF);
    end;
  until c = 'x';
end.

```

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.

```

program savesettings;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  yocto_api;

const
  serial = 'YCTOPOC1-123456'; // use serial number or logical name

var
  module : TYModule;
  errmsg  : string;
  newname : string;

begin
  // Setup the API to use local USB devices
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
    Write('RegisterHub error: '+errmsg);
    exit;
  end;

  module := yFindModule(serial);
  if (not(module.isOnline)) then
  begin
    writeln('Module not connected (check identification and USB cable)');
    exit;
  end;

```

```

end;

Writeln('Current logical name : '+module.get_logicalName());
Write('Enter new name : ');
Readln(newname);
if (not(yCheckLogicalName(newname))) then
begin
    Writeln('invalid logical name');
    exit;
end;
module.set_logicalName(newname);
module.saveToFlash();

Writeln('logical name is now : '+module.get_logicalName());
end.

```

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 `nil`. Below a short example listing the connected modules.

```

program inventory;
{$APPTYPE CONSOLE}
uses
    SysUtils,
    yocto_api;

var
    module : TYModule;
    errmsg : string;

begin
    // Setup the API to use local USB devices
    if yRegisterHub('usb', errmsg) <> YAPI_SUCCESS then
    begin
        Write('RegisterHub error: '+errmsg);
        exit;
    end;

    Writeln('Device list');

    module := yFirstModule();
    while module <> nil do
    begin
        Writeln( module.get_serialNumber()+' ('+module.get_productName()+')');
        module := module.nextModule();
    end;
end.

```

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

Global functions

yCheckLogicalName(name)

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

yDisableExceptions()

Disables the use of exceptions to report runtime errors.

yEnableExceptions()

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

yEnableUSBHost(osContext)

This function is used only on Android.

yFreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

yGetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

yGetTickCount()

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

yHandleEvents(errmsg)

Maintains the device-to-library communication channel.

yInitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

yPreregisterHub(url, errmsg)

Fault-tolerant alternative to `RegisterHub()`.

yRegisterDeviceArrivalCallback(arrivalCallback)

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

yRegisterDeviceRemovalCallback(removalCallback)

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

yRegisterHub(url, errmsg)

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

yRegisterHubDiscoveryCallback(hubDiscoveryCallback)

3. Reference

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

yRegisterLogFunction(logfun)

Registers a log callback function.

ySelectArchitecture(arch)

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

ySetDelegate(object)

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

ySetTimeout(callback, ms_timeout, arguments)

Invoke the specified callback function after a given timeout.

ySleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

yTriggerHubDiscovery(errmsg)

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

yUnregisterHub(url)

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

yUpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

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

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

```
function yCheckLogicalName( name: string): boolean
```

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

Parameters :

name a string containing the name to check.

Returns :

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

YAPI.DisableExceptions()

YAPI

yDisableExceptions()**yDisableExceptions()**

Disables the use of exceptions to report runtime errors.

procedure **yDisableExceptions()**

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

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

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

procedure **yEnableExceptions**()

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

YAPI.FreeAPI() yFreeAPI()yFreeAPI()

YAPI

Frees dynamically allocated memory blocks used by the Yoctopuce library.

procedure **yFreeAPI**()

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

YAPI.GetAPIVersion() yGetAPIVersion()yGetAPIVersion()

YAPI

Returns the version identifier for the Yoctopuce library in use.

```
function yGetAPIVersion( ): string
```

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

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

Returns :

a character string describing the library version.

YAPI.GetTickCount()

YAPI

yGetTickCount()yGetTickCount()

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

```
function yGetTickCount( ): u64
```

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

Returns :

a long integer corresponding to the millisecond counter.

YAPI.HandleEvents() yHandleEvents()yHandleEvents()

YAPI

Maintains the device-to-library communication channel.

```
function yHandleEvents( var errmsg: string): integer
```

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

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

Parameters :

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

Returns :

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

YAPI.InitAPI() yInitAPI()/yInitAPI()

YAPI

Initializes the Yoctopuce programming library explicitly.

```
function yInitAPI( mode: integer, var errmsg: string): integer
```

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

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

Parameters :

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

Returns :

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

YAPI.PreregisterHub() yPreregisterHub()yPreregisterHub()

YAPI

Fault-tolerant alternative to RegisterHub().

```
function yPreregisterHub( url: string, var errmsg: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterDeviceArrivalCallback()
yRegisterDeviceArrivalCallback()
yRegisterDeviceArrivalCallback()

YAPI

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

```
procedure yRegisterDeviceArrivalCallback( arrivalCallback: yDeviceUpdateFunc)
```

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

Parameters :

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

**YAPI.RegisterDeviceRemovalCallback()
yRegisterDeviceRemovalCallback()
yRegisterDeviceRemovalCallback()**

YAPI

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

```
procedure yRegisterDeviceRemovalCallback( removalCallback: yDeviceUpdateFunc)
```

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

Parameters :

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

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

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

```
function yRegisterHub( url: string, var errmsg: string): integer
```

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

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

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

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

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

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

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

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterHubDiscoveryCallback()
yRegisterHubDiscoveryCallback()
yRegisterHubDiscoveryCallback()

YAPI

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

```
procedure yRegisterHubDiscoveryCallback( hubDiscoveryCallback: YHubDiscoveryCallback)
```

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

Parameters :

hubDiscoveryCallback a procedure taking two string parameter, or null

YAPI.RegisterLogFunction()

YAPI

yRegisterLogFunction()yRegisterLogFunction()

Registers a log callback function.

```
procedure yRegisterLogFunction( logfun: yLogFunc)
```

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

Parameters :

logfun a procedure taking a string parameter, or null

YAPI.Sleep() ySleep()ySleep()

YAPI

Pauses the execution flow for a specified duration.

```
function ySleep( ms_duration: integer, var errmsg: string): integer
```

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

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

Parameters :

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

Returns :

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

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

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.

```
function yTriggerHubDiscovery( var errmsg: string): integer
```

Parameters :

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

Returns :

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

YAPI.UnregisterHub() yUnregisterHub()yUnregisterHub()

YAPI

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

procedure **yUnregisterHub**(**url**: string)

Parameters :

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

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

Triggers a (re)detection of connected Yoctopuce modules.

```
function yUpdateDeviceList( var errmsg: string): integer
```

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

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

Parameters :

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

Returns :

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

3.2. Accelerometer function interface

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

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

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

Global functions

yFindAccelerometer(func)

Retrieves an accelerometer for a given identifier.

yFirstAccelerometer()

Starts the enumeration of accelerometers currently accessible.

YAccelerometer methods

accelerometer→calibrateFromPoints(rawValues, refValues)

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

accelerometer→describe()

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

accelerometer→get_advertisedValue()

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

accelerometer→get_currentRawValue()

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

accelerometer→get_currentValue()

Returns the current value of the acceleration.

accelerometer→get_errorMessage()

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

accelerometer→get_errorType()

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

accelerometer→get_friendlyName()

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

accelerometer→get_functionDescriptor()

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

accelerometer→get_functionId()

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

accelerometer→get_hardwareId()

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

accelerometer→get_highestValue()

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

accelerometer→get_logFrequency()

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

accelerometer→get_logicalName()

Returns the logical name of the accelerometer.

accelerometer→get_lowestValue()

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

accelerometer→get_module()

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

accelerometer→get_module_async(callback, context)

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

accelerometer→get_recordedData(startTime, endTime)

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

accelerometer→get_reportFrequency()

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

accelerometer→get_resolution()

Returns the resolution of the measured values.

accelerometer→get_unit()

Returns the measuring unit for the acceleration.

accelerometer→get_userData()

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

accelerometer→get_xValue()

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

accelerometer→get_yValue()

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

accelerometer→get_zValue()

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

accelerometer→isOnline()

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

accelerometer→isOnline_async(callback, context)

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

accelerometer→load(msValidity)

Preloads the accelerometer cache with a specified validity duration.

accelerometer→loadCalibrationPoints(rawValues, refValues)

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

accelerometer→load_async(msValidity, callback, context)

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

accelerometer→nextAccelerometer()

Continues the enumeration of accelerometers started using yFirstAccelerometer().

accelerometer→registerTimedReportCallback(callback)

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

accelerometer→registerValueCallback(callback)

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

accelerometer→set_highestValue(newval)

Changes the recorded maximal value observed.

accelerometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

accelerometer→set_logicalName(newval)

Changes the logical name of the accelerometer.

accelerometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

accelerometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

accelerometer→set_resolution(newval)

Changes the resolution of the measured physical values.

accelerometer→set_userData(data)

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

accelerometer→wait_async(callback, context)

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

YAccelerometer.FindAccelerometer() yFindAccelerometer(yFindAccelerometer())

YAccelerometer

Retrieves an accelerometer for a given identifier.

```
function yFindAccelerometer( func: string): TYAccelerometer
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the accelerometer

Returns :

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

YAccelerometer.FirstAccelerometer() yFirstAccelerometer()yFirstAccelerometer()

YAccelerometer

Starts the enumeration of accelerometers currently accessible.

```
function yFirstAccelerometer( ): TYAccelerometer
```

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

Returns :

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

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

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

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                                refValues: TDoubleArray): LongInt
```

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→describe()accelerometer.describe()**YAccelerometer**

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

```
function describe( ): string
```

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

Returns :

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

accelerometer→get_advertisedValue()

YAccelerometer

accelerometer→advertisedValue()

accelerometer.get_advertisedValue()

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

```
function get_advertisedValue( ): string
```

Returns :

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

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

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

```
function get_currentRawValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

accelerometer→get_currentValue()

YAccelerometer

accelerometer→currentValue()

accelerometer.get_currentValue()

Returns the current value of the acceleration.

```
function get_currentValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

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

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

```
function get_errorMessage( ): string
```

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

Returns :

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

accelerometer→get_errorType()
accelerometer→errorType()
accelerometer.get_errorType()

YAccelerometer

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

```
function get_errorType( ): YRETCODE
```

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

Returns :

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

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

YAccelerometer

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

```
function get_functionDescriptor( ): YFUN_DESCR
```

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

Returns :

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

accelerometer→get_highestValue()

YAccelerometer

accelerometer→highestValue()

accelerometer.get_highestValue()

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

```
function get_highestValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

accelerometer→get_logFrequency()**YAccelerometer****accelerometer→logFrequency()****accelerometer.get_logFrequency()**

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

```
function get_logFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

accelerometer→get_logicalName()

YAccelerometer

accelerometer→logicalName()

accelerometer.get_logicalName()

Returns the logical name of the accelerometer.

```
function get_logicalName( ): string
```

Returns :

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

accelerometer→get_lowestValue()**YAccelerometer****accelerometer→lowestValue()****accelerometer.get_lowestValue()**

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

```
function get_lowestValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

accelerometer→**get_module()**

YAccelerometer

accelerometer→**module()****accelerometer.get_module()**

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

```
function get_module( ): YModule
```

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

Returns :

an instance of YModule

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

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

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

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

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

Parameters :

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

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

Returns :

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

accelerometer→get_reportFrequency()

YAccelerometer

accelerometer→reportFrequency()

accelerometer.get_reportFrequency()

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

```
function get_reportFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

accelerometer→get_resolution()**YAccelerometer****accelerometer→resolution()****accelerometer.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

accelerometer→**get_unit()**

YAccelerometer

accelerometer→**unit()****accelerometer.get_unit()**

Returns the measuring unit for the acceleration.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the acceleration

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

accelerometer→get_userData()**YAccelerometer****accelerometer→userData()****accelerometer.get_userData()**

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

```
function get_userData( ): Tobject
```

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

Returns :

the object stored previously by the caller.

accelerometer→**get_xValue()**

YAccelerometer

accelerometer→**xValue()****accelerometer.get_xValue()**

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

```
function get_xValue( ): double
```

Returns :

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

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

accelerometer→get_yValue()**YAccelerometer****accelerometer→yValue()accelerometer.get_yValue()**

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

function **get_yValue()**: double

Returns :

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

On failure, throws an exception or returns Y_YVALUE_INVALID.

accelerometer→get_zValue()

YAccelerometer

accelerometer→zValue()accelerometer.get_zValue()

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

```
function get_zValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_ZVALUE_INVALID.

accelerometer→isOnline()accelerometer.isOnline()**YAccelerometer**

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

```
function isOnline(): boolean
```

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

Returns :

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

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

Preloads the accelerometer cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

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

Parameters :

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

Returns :

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

**accelerometer→loadCalibrationPoints()
accelerometer.loadCalibrationPoints()**

YAccelerometer

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

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→**nextAccelerometer()**
accelerometer.nextAccelerometer()

YAccelerometer

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

```
function nextAccelerometer( ): TYAccelerometer
```

Returns :

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

accelerometer→registerTimedReportCallback()
accelerometer.registerTimedReportCallback()

YAccelerometer

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

```
function registerTimedReportCallback( callback: TYAccelerometerTimedReportCallback): LongInt
```

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

Parameters :

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

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

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

```
function registerValueCallback( callback: TYAccelerometerValueCallback): LongInt
```

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

Parameters :

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

accelerometer→set_highestValue()**YAccelerometer****accelerometer→setHighestValue()****accelerometer.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_logFrequency()

YAccelerometer

accelerometer→setLogFrequency()

accelerometer.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_logicalName()**YAccelerometer****accelerometer→setLogicalName()****accelerometer.set_logicalName()**

Changes the logical name of the accelerometer.

```
function set_logicalName( newval: string): integer
```

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

Parameters :

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

Returns :

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

accelerometer→set_lowestValue()
accelerometer→setLowestValue()
accelerometer.set_lowestValue()

YAccelerometer

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_reportFrequency()
accelerometer→setReportFrequency()
accelerometer.set_reportFrequency()

YAccelerometer

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_resolution()
accelerometer→setResolution()
accelerometer.set_resolution()

YAccelerometer

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_userdata()
accelerometer→setUserData()
accelerometer.set_userdata()

YAccelerometer

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

```
procedure set_userdata( data: Tobject)
```

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

Parameters :

data any kind of object to be stored

3.3. AnButton function interface

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

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

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

Global functions

yFindAnButton(func)

Retrieves an analog input for a given identifier.

yFirstAnButton()

Starts the enumeration of analog inputs currently accessible.

YAnButton methods

anbutton→describe()

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

anbutton→get_advertisedValue()

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

anbutton→get_analogCalibration()

Tells if a calibration process is currently ongoing.

anbutton→get_calibratedValue()

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

anbutton→get_calibrationMax()

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

anbutton→get_calibrationMin()

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

anbutton→get_errorMessage()

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

anbutton→get_errorType()

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

anbutton→get_friendlyName()

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

anbutton→get_functionDescriptor()

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

anbutton→get_functionId()

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

anbutton→get_hardwareId()

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

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

anbutton→get_module_async(callback, context)

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

anbutton→get_pulseCounter()

Returns the pulse counter value

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms)

anbutton→get_rawValue()

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

anbutton→get_sensitivity()

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

anbutton→get_userData()

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

anbutton→isOnline()

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

anbutton→isOnline_async(callback, context)

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

anbutton→load(msValidity)

Preloads the analog input cache with a specified validity duration.

anbutton→load_async(msValidity, callback, context)

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

anbutton→nextAnButton()

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

anbutton→registerValueCallback(callback)

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

anbutton→resetCounter()

Returns the pulse counter value as well as his timer

anbutton→set_analogCalibration(newval)

Starts or stops the calibration process.

anbutton→set_calibrationMax(newval)

3. Reference

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

anbutton→**set_calibrationMin**(newval)

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

anbutton→**set_logicalName**(newval)

Changes the logical name of the analog input.

anbutton→**set_sensitivity**(newval)

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

anbutton→**set_userData**(data)

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

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

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

YAnButton.FindAnButton() yFindAnButton()yFindAnButton()

YAnButton

Retrieves an analog input for a given identifier.

```
function yFindAnButton( func: string): TYAnButton
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the analog input

Returns :

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

YAnButton.FirstAnButton() yFirstAnButton()yFirstAnButton()

YAnButton

Starts the enumeration of analog inputs currently accessible.

```
function yFirstAnButton( ): TYAnButton
```

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

Returns :

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

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

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

```
function describe( ): string
```

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

Returns :

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

anbutton→**get_advertisedValue()**

YAnButton

anbutton→**advertisedValue()**

anbutton.get_advertisedValue()

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

```
function get_advertisedValue( ): string
```

Returns :

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

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

YAnButton

Tells if a calibration process is currently ongoing.

```
function get_analogCalibration( ): Integer
```

Returns :

either Y_ANALOGCALIBRATION_OFF or Y_ANALOGCALIBRATION_ON

On failure, throws an exception or returns Y_ANALOGCALIBRATION_INVALID.

anbutton→**get_calibratedValue()**

YAnButton

anbutton→**calibratedValue()**

anbutton.get_calibratedValue()

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

```
function get_calibratedValue( ): LongInt
```

Returns :

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

On failure, throws an exception or returns Y_CALIBRATEDVALUE_INVALID.

anbutton→get_calibrationMax()
anbutton→calibrationMax()
anbutton.get_calibrationMax()

YAnButton

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

```
function get_calibrationMax( ): LongInt
```

Returns :

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

On failure, throws an exception or returns Y_CALIBRATIONMAX_INVALID.

anbutton→**get_calibrationMin()**
anbutton→**calibrationMin()**
anbutton.get_calibrationMin()

YAnButton

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

```
function get_calibrationMin( ): LongInt
```

Returns :

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

On failure, throws an exception or returns Y_CALIBRATIONMIN_INVALID.

anbutton→get_errorMessage()
anbutton→errorMessage()
anbutton.get_errorMessage()

YAnButton

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

```
function get_errorMessage( ): string
```

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

Returns :

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

anbutton→**get_errorType()**

YAnButton

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

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

```
function get_errorType( ): YRETCODE
```

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

Returns :

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

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

YAnButton

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

```
function get_functionDescriptor( ): YFUN_DESCR
```

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

Returns :

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

anbutton→get_isPressed()

YAnButton

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

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

function **get_isPressed()**: Integer

Returns :

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

On failure, throws an exception or returns Y_ISPRESSED_INVALID.

anbutton→**get_lastTimePressed()**
anbutton→**lastTimePressed()**
anbutton.get_lastTimePressed()

YAnButton

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

```
function get_lastTimePressed( ): int64
```

Returns :

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

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

anbutton→get_lastTimeReleased()

YAnButton

anbutton→lastTimeReleased()

anbutton.get_lastTimeReleased()

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

```
function get_lastTimeReleased( ): int64
```

Returns :

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

On failure, throws an exception or returns Y_LASTTIMERELASED_INVALID.

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

Returns the logical name of the analog input.

```
function get_logicalName( ): string
```

Returns :

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

anbutton→**get_module()**

YAnButton

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

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

```
function get_module( ): TYModule
```

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

Returns :

an instance of YModule

anbutton→get_pulseCounter()
anbutton→pulseCounter()
anbutton.get_pulseCounter()

YAnButton

Returns the pulse counter value

```
function get_pulseCounter( ): int64
```

Returns :

an integer corresponding to the pulse counter value

On failure, throws an exception or returns Y_PULSECOUNTER_INVALID.

anbutton→**get_pulseTimer()**

YAnButton

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

Returns the timer of the pulses counter (ms)

```
function get_pulseTimer( ): int64
```

Returns :

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

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

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

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

function **get_rawValue()**: LongInt

Returns :

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

On failure, throws an exception or returns Y_RAWVALUE_INVALID.

anbutton→**get_sensitivity()**

YAnButton

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

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

function **get_sensitivity**(): LongInt

Returns :

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

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

anbutton→get_userData()**YAnButton****anbutton→userData()anbutton.get_userData()**

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

function **get_userData()**: Tobject

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

Returns :

the object stored previously by the caller.

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

YAnButton

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

```
function isOnline( ): boolean
```

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

Returns :

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

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

Preloads the analog input cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

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

Parameters :

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

Returns :

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

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

YAnButton

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

```
function nextAnButton( ): TYAnButton
```

Returns :

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

anbutton→registerValueCallback()
anbutton.registerValueCallback()

YAnButton

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

```
function registerValueCallback( callback: TYAnButtonValueCallback): LongInt
```

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

Parameters :

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

anbutton→**resetCounter()****anbutton.resetCounter()**

YAnButton

Returns the pulse counter value as well as his timer

function **resetCounter**(): LongInt

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→**set_analogCalibration()**
anbutton→**setAnalogCalibration()**
anbutton.set_analogCalibration()

YAnButton

Starts or stops the calibration process.

```
function set_analogCalibration( newval: Integer): integer
```

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

Parameters :

newval either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

anbutton→set_calibrationMax()

YAnButton

anbutton→setCalibrationMax()

anbutton.set_calibrationMax()

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

```
function set_calibrationMax( newval: LongInt): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→set_calibrationMin()
anbutton→setCalibrationMin()
anbutton.set_calibrationMin()

YAnButton

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

```
function set_calibrationMin( newval: LongInt): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the analog input.

```
function set_logicalName( newval: string): integer
```

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

Parameters :

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

Returns :

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

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

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

```
function set_sensitivity( newval: LongInt): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

anbutton→**set_userData()**

YAnButton

anbutton→**setUserData()****anbutton.set_userData()**

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

```
procedure set_userData( data: Tobject)
```

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

Parameters :

data any kind of object to be stored

3.4. CarbonDioxide function interface

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

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

js	<script type='text/javascript' src='yocto_carbondioxide.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YCarbonDioxide = yoctolib.YCarbonDioxide;
php	require_once('yocto_carbondioxide.php');
c++	#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.

carbondioxide→get_currentValue()

Returns the current value of the CO2 concentration.

carbondioxide→get_errorMessage()

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

carbondioxide→get_errorType()

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

carbondioxide→get_friendlyName()

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

carbondioxide→get_functionDescriptor()

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

carbondioxide→get_functionId()

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

carbondioxide→get_hardwareId()

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

carbondioxide→get_highestValue()

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

carbondioxide→get_logFrequency()

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

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

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

carbondioxide→get_module()

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

carbondioxide→get_module_async(callback, context)

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

carbondioxide→get_recordedData(startTime, endTime)

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

carbondioxide→get_reportFrequency()

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_unit()

Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

carbondioxide→isOnline_async(callback, context)

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

carbondioxide→load(msValidity)

Preloads the CO2 sensor cache with a specified validity duration.

carbondioxide→loadCalibrationPoints(rawValues, refValues)

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

carbondioxide→load_async(msValidity, callback, context)

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

carbondioxide→nextCarbonDioxide()

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

carbondioxide→registerTimedReportCallback(callback)

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

carbondioxide→registerValueCallback(callback)

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

carbondioxide→set_highestValue(newval)

Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(newval)

Changes the logical name of the CO2 sensor.

carbondioxide→set_lowestValue(newval)

Changes the recorded minimal value observed.

carbondioxide→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

carbondioxide→set_resolution(newval)

Changes the resolution of the measured physical values.

carbondioxide→set_userData(data)

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

carbondioxide→wait_async(callback, context)

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

YCarbonDioxide.FindCarbonDioxide() yFindCarbonDioxide()yFindCarbonDioxide()

YCarbonDioxide

Retrieves a CO2 sensor for a given identifier.

```
function yFindCarbonDioxide( func: string): TYCarbonDioxide
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the CO2 sensor

Returns :

a YCarbonDioxide object allowing you to drive the CO2 sensor.

YCarbonDioxide.FirstCarbonDioxide()
yFirstCarbonDioxide()**yFirstCarbonDioxide()**

YCarbonDioxide

Starts the enumeration of CO2 sensors currently accessible.

```
function yFirstCarbonDioxide( ): TYCarbonDioxide
```

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

Returns :

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

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

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

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→describe()carbondioxide.describe()**YCarbonDioxide**

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

```
function describe( ): string
```

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

Returns :

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

carbondioxide→get_advertisedValue()

YCarbonDioxide

carbondioxide→advertisedValue()

carbondioxide.get_advertisedValue()

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

```
function get_advertisedValue( ): string
```

Returns :

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

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

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

```
function get_currentRawValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

carbondioxide→get_currentValue()
carbondioxide→currentValue()
carbondioxide.get_currentValue()

YCarbonDioxide

Returns the current value of the CO2 concentration.

```
function get_currentValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

carbondioxide→get_errorMessage()
carbondioxide→errorMessage()
carbondioxide.get_errorMessage()

YCarbonDioxide

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

```
function get_errorMessage( ): string
```

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

Returns :

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

carbondioxide→get_errorType()
carbondioxide→errorType()
carbondioxide.get_errorType()

YCarbonDioxide

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

```
function get_errorType( ): YRETCODE
```

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

Returns :

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

carbondioxide→get_functionDescriptor()
carbondioxide→functionDescriptor()
carbondioxide.get_functionDescriptor()

YCarbonDioxide

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

```
function get_functionDescriptor( ): YFUN_DESCR
```

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

Returns :

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

carbondioxide→get_highestValue()
carbondioxide→highestValue()
carbondioxide.get_highestValue()

YCarbonDioxide

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

```
function get_highestValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

carbondioxide→get_logFrequency()
carbondioxide→logFrequency()
carbondioxide.get_logFrequency()

YCarbonDioxide

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

```
function get_logFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

carbondioxide→get_logicalName()
carbondioxide→logicalName()
carbondioxide.get_logicalName()

YCarbonDioxide

Returns the logical name of the CO2 sensor.

```
function get_logicalName( ): string
```

Returns :

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

carbondioxide→get_lowestValue()
carbondioxide→lowestValue()
carbondioxide.get_lowestValue()

YCarbonDioxide

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

```
function get_lowestValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

carbondioxide→get_module()
carbondioxide→module()
carbondioxide.get_module()

YCarbonDioxide

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

```
function get_module( ): TModule
```

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

Returns :

an instance of YModule

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

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

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

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

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

Parameters :

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

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

Returns :

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

carbondioxide→get_reportFrequency()

YCarbonDioxide

carbondioxide→reportFrequency()

carbondioxide.get_reportFrequency()

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

```
function get_reportFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

carbondioxide→get_resolution()**YCarbonDioxide****carbondioxide→resolution()****carbondioxide.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

carbondioxide→**get_unit()**

YCarbonDioxide

carbondioxide→**unit()****carbondioxide.get_unit()**

Returns the measuring unit for the CO2 concentration.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the CO2 concentration

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

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

YCarbonDioxide

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

```
function get_userData( ): Tobject
```

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

Returns :

the object stored previously by the caller.

carbondioxide→isOnline()carbondioxide.isOnline()

YCarbonDioxide

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

```
function isOnline( ): boolean
```

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

Returns :

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

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

Preloads the CO2 sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

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

Parameters :

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

Returns :

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

carbondioxide→loadCalibrationPoints() carbondioxide.loadCalibrationPoints()

YCarbonDioxide

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

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→nextCarbonDioxide()
carbondioxide.nextCarbonDioxide()

YCarbonDioxide

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

```
function nextCarbonDioxide( ): TYCarbonDioxide
```

Returns :

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

**carbondioxide→registerTimedReportCallback()
carbondioxide.registerTimedReportCallback()**

YCarbonDioxide

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

```
function registerTimedReportCallback( callback: TYCarbonDioxideTimedReportCallback): LongInt
```

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

Parameters :

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

**carbondioxide→registerValueCallback()
carbondioxide.registerValueCallback()**

YCarbonDioxide

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

```
function registerValueCallback( callback: TYCarbonDioxideValueCallback): LongInt
```

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

Parameters :

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

carbondioxide→set_highestValue()
carbondioxide→setHighestValue()
carbondioxide.set_highestValue()

YCarbonDioxide

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logFrequency()
carbondioxide→setLogFrequency()
carbondioxide.set_logFrequency()

YCarbonDioxide

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logicalName()
carbondioxide→setLogicalName()
carbondioxide.set_logicalName()

YCarbonDioxide

Changes the logical name of the CO2 sensor.

```
function set_logicalName( newval: string): integer
```

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

Parameters :

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

Returns :

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

carbondioxide→set_lowestValue()
carbondioxide→setLowestValue()
carbondioxide.set_lowestValue()

YCarbonDioxide

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_reportFrequency()
carbondioxide→setReportFrequency()
carbondioxide.set_reportFrequency()

YCarbonDioxide

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_resolution()
carbondioxide→setResolution()
carbondioxide.set_resolution()

YCarbonDioxide

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_userdata()
carbondioxide→setUserData()
carbondioxide.set_userdata()

YCarbonDioxide

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

```
procedure set_userdata( data: Tobject)
```

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

Parameters :

data any kind of object to be stored

3.5. ColorLed function interface

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

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

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

Global functions

yFindColorLed(func)

Retrieves an RGB led for a given identifier.

yFirstColorLed()

Starts the enumeration of RGB leds currently accessible.

YColorLed methods

colorled→describe()

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

colorled→get_advertisedValue()

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

colorled→get_errorMessage()

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

colorled→get_errorType()

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

colorled→get_friendlyName()

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

colorled→get_functionDescriptor()

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

colorled→get_functionId()

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

colorled→get_hardwareId()

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

colorled→get_hslColor()

Returns the current HSL color of the led.

colorled→get_logicalName()

Returns the logical name of the RGB led.

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.

YColorLed.FindColorLed() yFindColorLed()yFindColorLed()

YColorLed

Retrieves an RGB led for a given identifier.

```
function yFindColorLed( func: string): TYColorLed
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the RGB led

Returns :

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

YColorLed.FirstColorLed()
yFirstColorLed()yFirstColorLed()

YColorLed

Starts the enumeration of RGB leds currently accessible.

```
function yFirstColorLed( ): TYColorLed
```

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

Returns :

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

colorled→describe()colorled.describe()**YColorLed**

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

```
function describe( ): string
```

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

Returns :

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

colorled→**get_advertisedValue()**

YColorLed

colorled→**advertisedValue()**

colorled.get_advertisedValue()

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

```
function get_advertisedValue( ): string
```

Returns :

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

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

YColorLed

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

```
function get_errorMessage( ): string
```

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

Returns :

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

colorled→**get_errorType()**

YColorLed

colorled→**errorType()****colorled.get_errorType()**

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

```
function get_errorType( ): YRETCODE
```

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

Returns :

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

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

YColorLed

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

```
function get_functionDescriptor( ): YFUN_DESCR
```

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

Returns :

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

colorled→**get_hslColor()**

YColorLed

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

Returns the current HSL color of the led.

```
function get_hslColor( ): LongInt
```

Returns :

an integer corresponding to the current HSL color of the led

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

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

Returns the logical name of the RGB led.

```
function get_logicalName( ): string
```

Returns :

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

colorled→**get_module()**

YColorLed

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

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

```
function get_module( ): TYModule
```

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

Returns :

an instance of YModule

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

Returns the current RGB color of the led.

```
function get_rgbColor( ): LongInt
```

Returns :

an integer corresponding to the current RGB color of the led

On failure, throws an exception or returns Y_RGBCOLOR_INVALID.

colorled→get_rgbColorAtPowerOn()
colorled→rgbColorAtPowerOn()
colorled.get_rgbColorAtPowerOn()

YColorLed

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

```
function get_rgbColorAtPowerOn( ): LongInt
```

Returns :

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

On failure, throws an exception or returns Y_RGBCOLORATPOWERON_INVALID.

colorled→**get_userData()****YColorLed****colorled**→**userData()****colorled.get_userData()**

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

function **get_userData**(): Tobject

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

Returns :

the object stored previously by the caller.

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

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

```
function hslMove( hsl_target: LongInt, ms_duration: LongInt): integer
```

Parameters :

hsl_target desired HSL color at the end of the transition
ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

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

```
function isOnline( ): boolean
```

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

Returns :

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

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

Preloads the RGB led cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

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

Parameters :

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

Returns :

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

colorled→**nextColorLed()****colorled.nextColorLed()****YColorLed**

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

```
function nextColorLed( ): TYColorLed
```

Returns :

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

colorled→registerValueCallback()
colorled.registerValueCallback()**YColorLed**

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

```
function registerValueCallback( callback: TYColorLedValueCallback): LongInt
```

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

Parameters :

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

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

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

```
function rgbMove( rgb_target: LongInt, ms_duration: LongInt): integer
```

Parameters :

rgb_target desired RGB color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_hslColor()**

YColorLed

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

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

```
function set_hslColor( newval: LongInt): integer
```

Encoding is done as follows: 0xHHSSL.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_logicalName()**
colorled→**setLogicalName()**
colorled.set_logicalName()

YColorLed

Changes the logical name of the RGB led.

```
function set_logicalName( newval: string): integer
```

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

Parameters :

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

Returns :

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

colorled→**set_rgbColor()**

YColorLed

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

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

```
function set_rgbColor( newval: LongInt): integer
```

Encoding is done as follows: 0xRRGGBB.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_rgbColorAtPowerOn()**
colorled→**setRgbColorAtPowerOn()**
colorled.set_rgbColorAtPowerOn()

YColorLed

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

```
function set_rgbColorAtPowerOn( newval: LongInt): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→**set_userdata()**

YColorLed

colorled→**setUserData()****colorled.set_userdata()**

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

```
procedure set_userdata( data: Tobject)
```

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

Parameters :

data any kind of object to be stored

3.6. Compass function interface

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

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

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

Global functions

yFindCompass(func)

Retrieves a compass for a given identifier.

yFirstCompass()

Starts the enumeration of compasses currently accessible.

YCompass methods

compass→calibrateFromPoints(rawValues, refValues)

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

compass→describe()

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

compass→get_advertisedValue()

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

compass→get_currentRawValue()

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

compass→get_currentValue()

Returns the current value of the relative bearing.

compass→get_errorMessage()

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

compass→get_errorType()

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

compass→get_friendlyName()

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

compass→get_functionDescriptor()

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

compass→get_functionId()

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

compass→get_hardwareId()

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

compass→get_highestValue()

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

compass→get_logFrequency()

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

compass→get_logicalName()

Returns the logical name of the compass.

compass→get_lowestValue()

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

compass→get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

compass→get_module()

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

compass→get_module_async(callback, context)

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

compass→get_recordedData(startTime, endTime)

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

compass→get_reportFrequency()

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

compass→get_resolution()

Returns the resolution of the measured values.

compass→get_unit()

Returns the measuring unit for the relative bearing.

compass→get_userData()

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

compass→isOnline()

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

compass→isOnline_async(callback, context)

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

compass→load(msValidity)

Preloads the compass cache with a specified validity duration.

compass→loadCalibrationPoints(rawValues, refValues)

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

compass→load_async(msValidity, callback, context)

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

compass→nextCompass()

Continues the enumeration of compasses started using yFirstCompass().

compass→registerTimedReportCallback(callback)

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

compass→registerValueCallback(callback)

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

compass→set_highestValue(newval)

Changes the recorded maximal value observed.

compass→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

compass→set_logicalName(newval)

Changes the logical name of the compass.

compass→set_lowestValue(newval)

Changes the recorded minimal value observed.

compass→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

compass→set_resolution(newval)

Changes the resolution of the measured physical values.

compass→set_userData(data)

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

compass→wait_async(callback, context)

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

YCompass.FindCompass() yFindCompass()yFindCompass()

YCompass

Retrieves a compass for a given identifier.

```
function yFindCompass( func: string): TYCompass
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the compass

Returns :

a YCompass object allowing you to drive the compass.

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

YCompass

Starts the enumeration of compasses currently accessible.

```
function yFirstCompass( ): TYCompass
```

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

Returns :

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

compass→calibrateFromPoints()
compass.calibrateFromPoints()**YCompass**

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

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                               refValues: TDoubleArray): LongInt
```

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→describe()compass.describe()**YCompass**

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

```
function describe( ): string
```

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

Returns :

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

compass→**get_advertisedValue()**

YCompass

compass→**advertisedValue()**

compass.get_advertisedValue()

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

```
function get_advertisedValue( ): string
```

Returns :

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

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

YCompass

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

```
function get_currentRawValue( ): double
```

Returns :

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

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

compass→**get_currentValue()**

YCompass

compass→**currentValue()**

compass.get_currentValue()

Returns the current value of the relative bearing.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the relative bearing

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

compass→**get_errorMessage()**
compass→**errorMessage()**
compass.get_errorMessage()

YCompass

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

```
function get_errorMessage( ): string
```

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

Returns :

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

compass→**get_errorType()**

YCompass

compass→**errorType()****compass.get_errorType()**

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

```
function get_errorType( ): YRETCODE
```

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

Returns :

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

compass→**get_functionDescriptor()**
compass→**functionDescriptor()**
compass.get_functionDescriptor()

YCompass

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

```
function get_functionDescriptor( ): YFUN_DESCR
```

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

Returns :

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

compass→**get_highestValue()**

YCompass

compass→**highestValue()**

compass.get_highestValue()

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

```
function get_highestValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

compass→get_logFrequency()**YCompass****compass→logFrequency()****compass.get_logFrequency()**

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

```
function get_logFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

compass→**get_logicalName()**

YCompass

compass→**logicalName()****compass.get_logicalName()**

Returns the logical name of the compass.

```
function get_logicalName( ): string
```

Returns :

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

compass→**get_lowestValue()****YCompass****compass**→**lowestValue()****compass.get_lowestValue()**

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

```
function get_lowestValue( ): double
```

Returns :

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

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

compass→**get_magneticHeading()**

YCompass

compass→**magneticHeading()**

compass.get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

```
function get_magneticHeading( ): double
```

Returns :

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

On failure, throws an exception or returns Y_MAGNETICHEADING_INVALID.

compass→**get_module()****YCompass****compass**→**module()****compass.get_module()**

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

```
function get_module( ): TYModule
```

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

Returns :

an instance of `YModule`

compass→get_recordedData()**YCompass****compass→recordedData()****compass.get_recordedData()**

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

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

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

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

Parameters :

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

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

Returns :

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

compass→get_reportFrequency()**YCompass****compass→reportFrequency()****compass.get_reportFrequency()**

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

```
function get_reportFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

compass→**get_resolution()**

YCompass

compass→**resolution()****compass.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

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

Returns :

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

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

compass→**get_unit()****YCompass****compass**→**unit()****compass.get_unit()**

Returns the measuring unit for the relative bearing.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the relative bearing

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

compass→**get_userData()**

YCompass

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

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

function **get_userData**(): Tobject

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

Returns :

the object stored previously by the caller.

compass→**isOnline()****compass.isOnline()****YCompass**

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

```
function isOnline( ): boolean
```

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

Returns :

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

compass→**load()****compass.load()****YCompass**

Preloads the compass cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

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

Parameters :

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

Returns :

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

compass→loadCalibrationPoints()
compass.loadCalibrationPoints()

YCompass

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

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**nextCompass()****compass.nextCompass()**

YCompass

Continues the enumeration of compasses started using `yFirstCompass()`.

```
function nextCompass( ): TYCompass
```

Returns :

a pointer to a `YCompass` object, corresponding to a compass currently online, or a `null` pointer if there are no more compasses to enumerate.

compass→registerTimedReportCallback()
compass.registerTimedReportCallback()

YCompass

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

```
function registerTimedReportCallback( callback: TYCompassTimedReportCallback): LongInt
```

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

Parameters :

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

compass→registerValueCallback()
compass.registerValueCallback()**YCompass**

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

```
function registerValueCallback( callback: TYCompassValueCallback): LongInt
```

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

Parameters :

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

compass→**set_highestValue()****YCompass****compass**→**setHighestValue()****compass.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_logFrequency()****YCompass****compass**→**setLogFrequency()****compass.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_logicalName()**
compass→**setLogicalName()**
compass.set_logicalName()

YCompass

Changes the logical name of the compass.

```
function set_logicalName( newval: string): integer
```

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

Parameters :

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

Returns :

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

compass→**set_lowestValue()****YCompass****compass**→**setLowestValue()****compass.set_lowestValue()**

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set_reportFrequency()
compass→setReportFrequency()
compass.set_reportFrequency()

YCompass

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_resolution()**

YCompass

compass→**setResolution()****compass.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→**set_userData()****YCompass****compass**→**setUserData()****compass.set_userData()**

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

procedure **set_userData**(**data**: Tobject)

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

Parameters :

data any kind of object to be stored

3.7. Current function interface

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

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

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

Global functions

yFindCurrent(func)

Retrieves a current sensor for a given identifier.

yFirstCurrent()

Starts the enumeration of current sensors currently accessible.

YCurrent methods

current→calibrateFromPoints(rawValues, refValues)

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

current→describe()

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

current→get_advertisedValue()

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

current→get_currentRawValue()

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

current→get_currentValue()

Returns the current measure for the current.

current→get_errorMessage()

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

current→get_errorType()

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

current→get_friendlyName()

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

current→get_functionDescriptor()

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

current→get_functionId()

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

current→get_hardwareId()

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

current→get_highestValue()

Returns the maximal value observed for the current.

current→get_logFrequency()

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

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

Returns the minimal value observed for the current.

current→get_module()

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

current→get_module_async(callback, context)

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

current→get_recordedData(startTime, endTime)

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

current→get_reportFrequency()

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

current→get_resolution()

Returns the resolution of the measured values.

current→get_unit()

Returns the measuring unit for the current.

current→get_userData()

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

current→isOnline()

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

current→isOnline_async(callback, context)

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

current→load(msValidity)

Preloads the current sensor cache with a specified validity duration.

current→loadCalibrationPoints(rawValues, refValues)

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

current→load_async(msValidity, callback, context)

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

current→nextCurrent()

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

current→registerTimedReportCallback(callback)

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

current→registerValueCallback(callback)

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

current→set_highestValue(newval)

Changes the recorded maximal value observed pour the current.

current→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

current→set_logicalName(newval)

Changes the logical name of the current sensor.

3. Reference

current→**set_lowestValue**(newval)

Changes the recorded minimal value observed pour the current.

current→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

current→**set_resolution**(newval)

Changes the resolution of the measured values.

current→**set_userData**(data)

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

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

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

YCurrent.FindCurrent() yFindCurrent()yFindCurrent()

YCurrent

Retrieves a current sensor for a given identifier.

```
function yFindCurrent( func: string): TYCurrent
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the current sensor

Returns :

a `YCurrent` object allowing you to drive the current sensor.

YCurrent.FirstCurrent()
yFirstCurrent()**yFirstCurrent()**

YCurrent

Starts the enumeration of current sensors currently accessible.

```
function yFirstCurrent( ): TYCurrent
```

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

Returns :

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

current→calibrateFromPoints()
current.calibrateFromPoints()

YCurrent

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

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→describe()current.describe()**YCurrent**

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

function **describe**(): string

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

Returns :

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

current→**get_advertisedValue()****YCurrent****current**→**advertisedValue()****current.get_advertisedValue()**

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

```
function get_advertisedValue( ): string
```

Returns :

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

current→**get_currentRawValue()**

YCurrent

current→**currentRawValue()**

current.get_currentRawValue()

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

```
function get_currentRawValue( ): double
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

current→**get_currentValue()****YCurrent****current**→**currentValue()****current.get_currentValue()**

Returns the current measure for the current.

function **get_currentValue**(): double

Returns :

a floating point number corresponding to the current measure for the current

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

current→**get_errorMessage()**

YCurrent

current→**errorMessage()****current.get_errorMessage()**

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

```
function get_errorMessage( ): string
```

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

Returns :

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

current→**get_errorType()****YCurrent****current**→**errorType()****current.get_errorType()**

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

```
function get_errorType( ): YRETCODE
```

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

Returns :

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

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

YCurrent

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

```
function get_functionDescriptor( ): YFUN_DESCR
```

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

Returns :

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

current→**get_highestValue()****YCurrent****current**→**highestValue()****current.get_highestValue()**

Returns the maximal value observed for the current.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the current

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

current→**get_logFrequency()**

YCurrent

current→**logFrequency()****current.get_logFrequency()**

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

```
function get_logFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

current→**get_logicalName()****YCurrent****current**→**logicalName()****current.get_logicalName()**

Returns the logical name of the current sensor.

```
function get_logicalName( ): string
```

Returns :

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

current→**get_lowestValue()**

YCurrent

current→**lowestValue()****current.get_lowestValue()**

Returns the minimal value observed for the current.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the current

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

current→**get_module()****YCurrent****current**→**module()****current.get_module()**

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

```
function get_module( ): TModule
```

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

Returns :

an instance of `YModule`

current→**get_recordedData()****YCurrent****current**→**recordedData()****current.get_recordedData()**

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

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

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

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

Parameters :

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

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

Returns :

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

current→get_reportFrequency()**YCurrent****current→reportFrequency()****current.get_reportFrequency()**

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

```
function get_reportFrequency( ): string
```

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

current→**get_resolution()**

YCurrent

current→**resolution()****current.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

current→**get_unit()****YCurrent****current**→**unit()****current.get_unit()**

Returns the measuring unit for the current.

function **get_unit**(): string

Returns :

a string corresponding to the measuring unit for the current

On failure, throws an exception or returns Y_UNIT_INVALID.

current→**get_userData()**

YCurrent

current→**userData()****current.get_userData()**

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

function **get_userData**(): Tobject

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

Returns :

the object stored previously by the caller.

current→isOnline()current.isOnline()**YCurrent**

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

```
function isOnline(): boolean
```

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

Returns :

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

current→load()current.load()**YCurrent**

Preloads the current sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

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

Parameters :

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

Returns :

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

current→loadCalibrationPoints()
current.loadCalibrationPoints()

YCurrent

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

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**nextCurrent()****current.nextCurrent()**

YCurrent

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

function **nextCurrent**(): TYCurrent

Returns :

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

current→registerTimedReportCallback()
current.registerTimedReportCallback()

YCurrent

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

```
function registerTimedReportCallback( callback: TYCurrentTimedReportCallback): LongInt
```

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

Parameters :

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

current→registerValueCallback()
current.registerValueCallback()**YCurrent**

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

```
function registerValueCallback( callback: TYCurrentValueCallback): LongInt
```

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

Parameters :

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

current→**set_highestValue()****YCurrent****current**→**setHighestValue()****current.set_highestValue()**

Changes the recorded maximal value observed pour the current.

```
function set_highestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_logFrequency()**

YCurrent

current→**setLogFrequency()**

current.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_logicalName()****YCurrent****current**→**setLogicalName()****current.set_logicalName()**

Changes the logical name of the current sensor.

```
function set_logicalName( newval: string): integer
```

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

Parameters :

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

Returns :

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

current→**set_lowestValue()**

YCurrent

current→**setLowestValue()****current.set_lowestValue()**

Changes the recorded minimal value observed pour the current.

```
function set_lowestValue( newval: double): integer
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_reportFrequency()**
current→**setReportFrequency()**
current.set_reportFrequency()

YCurrent

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_resolution()**

YCurrent

current→**setResolution()****current.set_resolution()**

Changes the resolution of the measured values.

```
function set_resolution( newval: double): integer
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→**set_userData()****YCurrent****current**→**setUserData()****current.set_userData()**

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

```
procedure set_userData( data: Tobject)
```

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

Parameters :

data any kind of object to be stored

3.8. DataLogger function interface

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

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

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

Global functions

yFindDataLogger(func)

Retrieves a data logger for a given identifier.

yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

YDataLogger methods

datalogger→describe()

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

datalogger→forgetAllDataStreams()

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

datalogger→get_advertisedValue()

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

datalogger→get_autoStart()

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

datalogger→get_currentRunIndex()

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

datalogger→get_dataSets()

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

datalogger→get_dataStreams(v)

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

datalogger→get_errorMessage()

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

datalogger→get_errorType()

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

datalogger→get_friendlyName()

Returns a global identifier of the data logger in the format MODULE_NAME . FUNCTION_NAME.

datalogger→get_functionDescriptor()

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

`datalogger→get_functionId()`

Returns the hardware identifier of the data logger, without reference to the module.

`datalogger→get_hardwareId()`

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

`datalogger→get_logicalName()`

Returns the logical name of the data logger.

`datalogger→get_module()`

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

`datalogger→get_module_async(callback, context)`

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

`datalogger→get_recording()`

Returns the current activation state of the data logger.

`datalogger→get_timeUTC()`

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

`datalogger→get_userData()`

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

`datalogger→isOnline()`

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

`datalogger→isOnline_async(callback, context)`

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

`datalogger→load(msValidity)`

Preloads the data logger cache with a specified validity duration.

`datalogger→load_async(msValidity, callback, context)`

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

`datalogger→nextDataLogger()`

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

`datalogger→registerValueCallback(callback)`

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

`datalogger→set_autoStart(newval)`

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

`datalogger→set_logicalName(newval)`

Changes the logical name of the data logger.

`datalogger→set_recording(newval)`

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

`datalogger→set_timeUTC(newval)`

Changes the current UTC time reference used for recorded data.

`datalogger→set_userData(data)`

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

`datalogger→wait_async(callback, context)`

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

YDataLogger.FindDataLogger() yFindDataLogger()yFindDataLogger()

YDataLogger

Retrieves a data logger for a given identifier.

```
function yFindDataLogger( func: string): TYDataLogger
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger() yFirstDataLogger()yFirstDataLogger()

YDataLogger

Starts the enumeration of data loggers currently accessible.

```
function yFirstDataLogger( ): TYDataLogger
```

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

Returns :

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

datalogger→describe()**datalogger.describe()****YDataLogger**

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

function **describe**(): string

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

Returns :

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

datalogger→forgetAllDataStreams()
datalogger.forgetAllDataStreams()

YDataLogger

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

```
function forgetAllDataStreams( ): LongInt
```

This method also resets the current run index to zero.

Returns :

YAPI_SUCCESS if the call succeeds.

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

datalogger→get_advertisedValue()

YDataLogger

datalogger→advertisedValue()

datalogger.get_advertisedValue()

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

```
function get_advertisedValue( ): string
```

Returns :

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

datalogger→**get_autoStart()****YDataLogger****datalogger**→**autoStart()****datalogger.get_autoStart()**

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

function **get_autoStart()**: Integer

Returns :

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

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

datalogger→get_currentRunIndex()**YDataLogger****datalogger→currentRunIndex()****datalogger.get_currentRunIndex()**

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

```
function get_currentRunIndex( ): LongInt
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRUNINDEX_INVALID.

datalogger→get_dataSets()**YDataLogger****datalogger→dataSets()datalogger.get_dataSets()**

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

```
function get_dataSets( ): TYDataSetArray
```

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

Returns :

a list of YDataSet object.

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

datalogger→get_dataStreams()
datalogger→dataStreams()
datalogger.get_dataStreams()

YDataLogger

Builds a list of all data streams hold by the data logger (legacy method).

```
function get_dataStreams( v: Tlist): integer
```

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

v an array of YDataStream objects to be filled in

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

`datalogger→get_errorMessage()`**YDataLogger****`datalogger→errorMessage()`****`datalogger.get_errorMessage()`**

Returns the error message of the latest error with the data logger.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the data logger object

datalogger→**get_errorType()**

YDataLogger

datalogger→**errorType()****datalogger.get_errorType()**

Returns the numerical error code of the latest error with the data logger.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the data logger object

datalogger→get_functionDescriptor()
datalogger→functionDescriptor()
datalogger.get_functionDescriptor()

YDataLogger

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

datalogger→get_logicalName()

YDataLogger

datalogger→logicalName()

datalogger.get_logicalName()

Returns the logical name of the data logger.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the data logger. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

datalogger→get_module()**YDataLogger****datalogger→module()datalogger.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

datalogger→**get_recording()**

YDataLogger

datalogger→**recording()****datalogger.get_recording()**

Returns the current activation state of the data logger.

function **get_recording**(): Integer

Returns :

either Y_RECORDING_OFF or Y_RECORDING_ON, according to the current activation state of the data logger

On failure, throws an exception or returns Y_RECORDING_INVALID.

datalogger→get_timeUTC()**YDataLogger****datalogger→timeUTC()datalogger.get_timeUTC()**

Returns the Unix timestamp for current UTC time, if known.

```
function get_timeUTC( ): int64
```

Returns :

an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns Y_TIMEUTC_INVALID.

datalogger→**get_userData()**

YDataLogger

datalogger→**userData()****datalogger.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

datalogger→isOnline()datalogger.isOnline()**YDataLogger**

Checks if the data logger is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

Returns :

`true` if the data logger can be reached, and `false` otherwise

datalogger→**load()****datalogger.load()****YDataLogger**

Preloads the data logger cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

datalogger→**nextDataLogger()**
datalogger.nextDataLogger()

YDataLogger

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

```
function nextDataLogger( ): TYDataLogger
```

Returns :

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

datalogger→registerValueCallback()
datalogger.registerValueCallback()**YDataLogger**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYDataLoggerValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

datalogger→**set_autoStart()****YDataLogger****datalogger**→**setAutoStart()****datalogger.set_autoStart()**

Changes the default activation state of the data logger on power up.

```
function set_autoStart( newval: Integer): integer
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the default activation state of the data logger on power up

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_logicalName()**YDataLogger****datalogger→setLogicalName()****datalogger.set_logicalName()**

Changes the logical name of the data logger.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the data logger.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

datalogger→set_recording()**YDataLogger****datalogger→setRecording()****datalogger.set_recording()**

Changes the activation state of the data logger to start/stop recording data.

```
function set_recording( newval: Integer): integer
```

Parameters :

newval either Y_RECORDING_OFF or Y_RECORDING_ON, according to the activation state of the data logger to start/stop recording data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→**set_timeUTC()**

YDataLogger

datalogger→**setTimeUTC()****datalogger.set_timeUTC()**

Changes the current UTC time reference used for recorded data.

```
function set_timeUTC( newval: int64): integer
```

Parameters :

newval an integer corresponding to the current UTC time reference used for recorded data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_userdata()**YDataLogger****datalogger→setUserData()datalogger.set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.9. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_datalogger.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDataLogger = yoctolib.YDataLogger;
php	require_once('yocto_datalogger.php');
c++	#include "yocto_datalogger.h"
m	#import "yocto_datalogger.h"
pas	uses yocto_datalogger;
vb	yocto_datalogger.vb
cs	yocto_datalogger.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_datalogger import *

YDataRun methods
datarun→get_averageValue(measureName, pos) Returns the average value of the measure observed at the specified time period.
datarun→get_duration() Returns the duration (in seconds) of the data run.
datarun→get_maxValue(measureName, pos) Returns the maximal value of the measure observed at the specified time period.
datarun→get_measureNames() Returns the names of the measures recorded by the data logger.
datarun→get_minValue(measureName, pos) Returns the minimal value of the measure observed at the specified time period.
datarun→get_startTimeUTC() Returns the start time of the data run, relative to the Jan 1, 1970.
datarun→get_valueCount() Returns the number of values accessible in this run, given the selected data samples interval.
datarun→get_valueInterval() Returns the number of seconds covered by each value in this run.
datarun→set_valueInterval(valueInterval) Changes the number of seconds covered by each value in this run.

datarun→get_startTimeUTC()
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).

3.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instantiated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YAPI = yoctolib.YAPI;</code> <code>var YModule = yoctolib.YModule;</code>
php	<code>require_once('yocto_api.php');</code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>

YDataSet methods

dataset→get_endTimeUTC()

Returns the end time of the dataset, relative to the Jan 1, 1970.

dataset→get_functionId()

Returns the hardware identifier of the function that performed the measure, without reference to the module.

dataset→get_hardwareId()

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

dataset→get_measures()

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

dataset→get_preview()

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

dataset→get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dataset→get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

dataset→get_summary()

Returns an YMeasure object which summarizes the whole DataSet.

dataset→get_unit()

Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→**get_endTimeUTC()****YDataSet****dataset**→**endTimeUTC()****dataset.get_endTimeUTC()**

Returns the end time of the dataset, relative to the Jan 1, 1970.

```
function get_endTimeUTC( ): int64
```

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

dataset→**get_functionId()****YDataSet****dataset**→**functionId()****dataset.get_functionId()**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

```
function get_functionId( ): string
```

For example `temperature1`.

Returns :

a string that identifies the function (ex: `temperature1`)

dataset→**get_hardwareId()**

YDataSet

dataset→**hardwareId()****dataset.get_hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

```
function get_hardwareId( ): string
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `THRMCPL1-123456.temperature1`)

Returns :

a string that uniquely identifies the function (ex: `THRMCPL1-123456.temperature1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

dataset→**get_measures()****YDataSet****dataset**→**measures()****dataset.get_measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

```
function get_measures( ): TYMeasureArray
```

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :

a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→**get_preview()**

YDataSet

dataset→**preview()****dataset.get_preview()**

Returns a condensed version of the measures that can be retrieved in this YDataSet, as a list of YMeasure objects.

```
function get_preview( ): TYMeasureArray
```

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→**get_progress()****YDataSet****dataset**→**progress()****dataset.get_progress()**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

```
function get_progress( ): LongInt
```

When the object is instanciated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :

an integer in the range 0 to 100 (percentage of completion).

dataset→**get_startTimeUTC()**

YDataSet

dataset→**startTimeUTC()****dataset.get_startTimeUTC()**

Returns the start time of the dataset, relative to the Jan 1, 1970.

```
function get_startTimeUTC( ): int64
```

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

dataset→**get_summary()****YDataSet****dataset**→**summary()****dataset.get_summary()**

Returns an YMeasure object which summarizes the whole DataSet.

```
function get_summary( ): TYMeasure
```

It includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

Returns :

an YMeasure object

dataset→**get_unit()**

YDataSet

dataset→**unit()****dataset.get_unit()**

Returns the measuring unit for the measured value.

```
function get_unit( ): string
```

Returns :

a string that represents a physical unit.

On failure, throws an exception or returns `Y_UNIT_INVALID`.

dataset→**loadMore()****dataset.loadMore()****YDataSet**

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

```
function loadMore( ): LongInt
```

Returns :

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

3.11. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
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()

Returns the average of all measures observed within this stream.

datastream→get_columnCount()

Returns the number of data columns present in this stream.

datastream→get_columnNames()

Returns the title (or meaning) of each data column present in this stream.

datastream→get_data(row, col)

Returns a single measure from the data stream, specified by its row and column index.

datastream→get_dataRows()

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

datastream→get_dataSamplesIntervalMs()

Returns the number of milliseconds between two consecutive rows of this data stream.

datastream→get_duration()

Returns the approximate duration of this stream, in seconds.

datastream→get_maxValue()

Returns the largest measure observed within this stream.

datastream→get_minValue()

Returns the smallest measure observed within this stream.

datastream→get_rowCount()

Returns the number of data rows present in this stream.

datastream→get_runIndex()

Returns the run index of the data stream.

datastream→get_startTime()

Returns the relative start time of the data stream, measured in seconds.

datastream→get_startTimeUTC()

Returns the start time of the data stream, relative to the Jan 1, 1970.

datastream→get_averageValue()

YDataStream

datastream→averageValue()

datastream.get_averageValue()

Returns the average of all measures observed within this stream.

```
function get_averageValue( ): double
```

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_columnCount()**YDataStream****datastream→columnCount()****datastream.get_columnCount()**

Returns the number of data columns present in this stream.

```
function get_columnCount( ): LongInt
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

datastream→get_columnNames()

YDataStream

datastream→columnNames()

datastream.get_columnNames()

Returns the title (or meaning) of each data column present in this stream.

```
function get_columnNames( ): TStringArray
```

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes `_min`, `_avg` and `_max` respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

a list containing as many strings as there are columns in the data stream.

On failure, throws an exception or returns an empty array.

datastream→**get_data()****YDataStream****datastream**→**data()****datastream.get_data()**

Returns a single measure from the data stream, specified by its row and column index.

```
function get_data( row: LongInt, col: LongInt): double
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Parameters :

row row index

col column index

Returns :

a floating-point number

On failure, throws an exception or returns `Y_DATA_INVALID`.

datastream→**get_dataRows()**

YDataStream

datastream→**dataRows()****datastream.get_dataRows()**

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

```
function get_dataRows( ): TDoubleArrayArray
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Returns :

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.

datastream→get_dataSamplesIntervalMs()**YDataStream****datastream→dataSamplesIntervalMs()****datastream.get_dataSamplesIntervalMs()**

Returns the number of milliseconds between two consecutive rows of this data stream.

```
function get_dataSamplesIntervalMs( ): LongInt
```

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

Returns :

an unsigned number corresponding to a number of milliseconds.

datastream→**get_duration()**

YDataStream

datastream→**duration()****datastream.get_duration()**

Returns the approximate duration of this stream, in seconds.

function **get_duration**(): LongInt

Returns :

the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

datastream→**get_maxValue()****YDataStream****datastream**→**maxValue()****datastream.get_maxValue()**

Returns the largest measure observed within this stream.

```
function get_maxValue( ): double
```

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→**get_minValue()**

YDataStream

datastream→**minValue()****datastream.get_minValue()**

Returns the smallest measure observed within this stream.

```
function get_minValue( ): double
```

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the smallest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_rowCount()**YDataStream****datastream→rowCount()datastream.getRowCount()**

Returns the number of data rows present in this stream.

```
function get_rowCount( ): LongInt
```

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

datastream→**get_runIndex()**

YDataStream

datastream→**runIndex()****datastream.get_runIndex()**

Returns the run index of the data stream.

```
function get_runIndex( ): LongInt
```

A run can be made of multiple datastreams, for different time intervals.

Returns :

an unsigned number corresponding to the run index.

datastream→get_startTime()**YDataStream****datastream→startTime()datastream.get_startTime()**

Returns the relative start time of the data stream, measured in seconds.

```
function get_startTime( ): LongInt
```

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `get_startTimeUTC()`.

Returns :

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

datastream→get_startTimeUTC()

YDataStream

datastream→startTimeUTC()

datastream.get_startTimeUTC()

Returns the start time of the data stream, relative to the Jan 1, 1970.

```
function get_startTimeUTC( ): int64
```

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data stream (i.e. Unix time representation of the absolute time).

3.12. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_digitalio.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDigitalIO = yoctolib.YDigitalIO;
php	require_once('yocto_digitalio.php');
c++	#include "yocto_digitalio.h"
m	#import "yocto_digitalio.h"
pas	uses yocto_digitalio;
vb	yocto_digitalio.vb
cs	yocto_digitalio.cs
java	import com.yoctopuce.YoctoAPI.YDigitalIO;
py	from yocto_digitalio import *

Global functions

yFindDigitalIO(func)

Retrieves a digital IO port for a given identifier.

yFirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

YDigitalIO methods

digitalio→delayedPulse(bitno, ms_delay, ms_duration)

Schedules a pulse on a single bit for a specified duration.

digitalio→describe()

Returns a short text that describes unambiguously the instance of the digital IO port in the form TYPE (NAME) = SERIAL . FUNCTIONID.

digitalio→get_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

digitalio→get_bitDirection(bitno)

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

digitalio→get_bitOpenDrain(bitno)

Returns the type of electrical interface of a single bit from the I/O port.

digitalio→get_bitPolarity(bitno)

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

digitalio→get_bitState(bitno)

Returns the state of a single bit of the I/O port.

digitalio→get_errorMessage()

Returns the error message of the latest error with the digital IO port.

digitalio→get_errorType()

Returns the numerical error code of the latest error with the digital IO port.

digitalio→get_friendlyName()

Returns a global identifier of the digital IO port in the format MODULE_NAME . FUNCTION_NAME.

digitalio→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

digitalio→get_functionId()

Returns the hardware identifier of the digital IO port, without reference to the module.

digitalio→get_hardwareId()

Returns the unique hardware identifier of the digital IO port in the form `SERIAL.FUNCTIONID`.

digitalio→get_logicalName()

Returns the logical name of the digital IO port.

digitalio→get_module()

Gets the `YModule` object for the device on which the function is located.

digitalio→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

digitalio→get_outputVoltage()

Returns the voltage source used to drive output bits.

digitalio→get_portDirection()

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→get_portOpenDrain()

Returns the electrical interface for each bit of the port.

digitalio→get_portPolarity()

Returns the polarity of all the bits of the port.

digitalio→get_portSize()

Returns the number of bits implemented in the I/O port.

digitalio→get_portState()

Returns the digital IO port state: bit 0 represents input 0, and so on.

digitalio→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

digitalio→isOnline()

Checks if the digital IO port is currently reachable, without raising any error.

digitalio→isOnline_async(callback, context)

Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

digitalio→load(msValidity)

Preloads the digital IO port cache with a specified validity duration.

digitalio→load_async(msValidity, callback, context)

Preloads the digital IO port cache with a specified validity duration (asynchronous version).

digitalio→nextDigitalIO()

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

digitalio→pulse(bitno, ms_duration)

Triggers a pulse on a single bit for a specified duration.

digitalio→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

digitalio→set_bitDirection(bitno, bitdirection)

Changes the direction of a single bit from the I/O port.

digitalio→set_bitOpenDrain(bitno, opendrain)

Changes the electrical interface of a single bit from the I/O port.

digitalio→set_bitPolarity(bitno, bitpolarity)

Changes the polarity of a single bit from the I/O port.

digitalio→**set_bitState**(**bitno**, **bitstate**)

Sets a single bit of the I/O port.

digitalio→**set_logicalName**(**newval**)

Changes the logical name of the digital IO port.

digitalio→**set_outputVoltage**(**newval**)

Changes the voltage source used to drive output bits.

digitalio→**set_portDirection**(**newval**)

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→**set_portOpenDrain**(**newval**)

Changes the electrical interface for each bit of the port.

digitalio→**set_portPolarity**(**newval**)

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

digitalio→**set_portState**(**newval**)

Changes the digital IO port state: bit 0 represents input 0, and so on.

digitalio→**set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

digitalio→**toggle_bitState**(**bitno**)

Reverts a single bit of the I/O port.

digitalio→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDigitalIO.FindDigitalIO() yFindDigitalIO()yFindDigitalIO()

YDigitalIO

Retrieves a digital IO port for a given identifier.

```
function yFindDigitalIO( func: string): TYDigitalIO
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the digital IO port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDigitalIO.isOnline()` to test if the digital IO port is indeed online at a given time. In case of ambiguity when looking for a digital IO port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the digital IO port

Returns :

a `YDigitalIO` object allowing you to drive the digital IO port.

YDigitalIO.FirstDigitalIO()
yFirstDigitalIO()

YDigitalIO

Starts the enumeration of digital IO ports currently accessible.

```
function yFirstDigitalIO( ): TYDigitalIO
```

Use the method `YDigitalIO.nextDigitalIO()` to iterate on next digital IO ports.

Returns :

a pointer to a `YDigitalIO` object, corresponding to the first digital IO port currently online, or a `null` pointer if there are none.

Schedules a pulse on a single bit for a specified duration.

```
function delayedPulse( bitno: LongInt,  
                        ms_delay: LongInt,  
                        ms_duration: LongInt): LongInt
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

- bitno** the bit number; lowest bit has index 0
- ms_delay** waiting time before the pulse, in milliseconds
- ms_duration** desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→describe()digitalio.describe()**YDigitalIO**

Returns a short text that describes unambiguously the instance of the digital IO port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the digital IO port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

digitalio→**get_advertisedValue()**

YDigitalIO

digitalio→**advertisedValue()**

digitalio.get_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the digital IO port (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

digitalio→**get_bitDirection()****YDigitalIO****digitalio**→**bitDirection()****digitalio.get_bitDirection()**

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

```
function get_bitDirection( bitno: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**get_bitOpenDrain()****YDigitalIO****digitalio**→**bitOpenDrain()****digitalio.get_bitOpenDrain()**

Returns the type of electrical interface of a single bit from the I/O port.

```
function get_bitOpenDrain( bitno: LongInt): LongInt
```

(0 means the bit is an input, 1 an output).

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

0 means the a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

On failure, throws an exception or returns a negative error code.

digitalio→**get_bitPolarity()****YDigitalIO****digitalio**→**bitPolarity()****digitalio.get_bitPolarity()**

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

```
function get_bitPolarity( bitno: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**get_bitState()**

YDigitalIO

digitalio→**bitState()****digitalio.get_bitState()**

Returns the state of a single bit of the I/O port.

```
function get_bitState( bitno: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

the bit state (0 or 1)

On failure, throws an exception or returns a negative error code.

digitalio→**get_errorMessage()**
digitalio→**errorMessage()**
digitalio.get_errorMessage()

YDigitalIO

Returns the error message of the latest error with the digital IO port.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the digital IO port object

digitalio→**get_errorType()**

YDigitalIO

digitalio→**errorType()****digitalio.get_errorType()**

Returns the numerical error code of the latest error with the digital IO port.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the digital IO port object

digitalio→**get_functionDescriptor()**
digitalio→**functionDescriptor()**
digitalio.get_functionDescriptor()

YDigitalIO

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

digitalio→**get_logicalName()**

YDigitalIO

digitalio→**logicalName()****digitalio.get_logicalName()**

Returns the logical name of the digital IO port.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the digital IO port. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

digitalio→**get_module()****YDigitalIO****digitalio**→**module()****digitalio.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

digitalio→get_outputVoltage()

YDigitalIO

digitalio→outputVoltage()

digitalio.get_outputVoltage()

Returns the voltage source used to drive output bits.

```
function get_outputVoltage( ): Integer
```

Returns :

a value among Y_OUTPUTVOLTAGE_USB_5V, Y_OUTPUTVOLTAGE_USB_3V and Y_OUTPUTVOLTAGE_EXT_V corresponding to the voltage source used to drive output bits

On failure, throws an exception or returns Y_OUTPUTVOLTAGE_INVALID.

digitalio→**get_portDirection()****YDigitalIO****digitalio**→**portDirection()****digitalio.get_portDirection()**

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

```
function get_portDirection( ): LongInt
```

Returns :

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

On failure, throws an exception or returns Y_PORTDIRECTION_INVALID.

digitalio→get_portOpenDrain()
digitalio→portOpenDrain()
digitalio.get_portOpenDrain()

YDigitalIO

Returns the electrical interface for each bit of the port.

```
function get_portOpenDrain( ): LongInt
```

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

Returns :

an integer corresponding to the electrical interface for each bit of the port

On failure, throws an exception or returns Y_PORTOPENDRAIN_INVALID.

digitalio→**get_portPolarity()****YDigitalIO****digitalio**→**portPolarity()****digitalio.get_portPolarity()**

Returns the polarity of all the bits of the port.

function **get_portPolarity**(): LongInt

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

Returns :

an integer corresponding to the polarity of all the bits of the port

On failure, throws an exception or returns Y_PORTPOLARITY_INVALID.

digitalio→**get_portSize()**

YDigitalIO

digitalio→**portSize()****digitalio.get_portSize()**

Returns the number of bits implemented in the I/O port.

function **get_portSize**(): LongInt

Returns :

an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns Y_PORTSIZE_INVALID.

digitalio→**get_portState()****YDigitalIO****digitalio**→**portState()****digitalio.get_portState()**

Returns the digital IO port state: bit 0 represents input 0, and so on.

function **get_portState**(): LongInt

Returns :

an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

On failure, throws an exception or returns Y_PORTSTATE_INVALID.

digitalio→**get_userData()**

YDigitalIO

digitalio→**userData()****digitalio.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

digitalio→**isOnline()****digitalio.isOnline()****YDigitalIO**

Checks if the digital IO port is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the digital IO port.

Returns :

`true` if the digital IO port can be reached, and `false` otherwise

Preloads the digital IO port cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

digitalio→**nextDigitalIO()****digitalio.nextDigitalIO()****YDigitalIO**

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

```
function nextDigitalIO( ): TYDigitalIO
```

Returns :

a pointer to a `YDigitalIO` object, corresponding to a digital IO port currently online, or a `null` pointer if there are no more digital IO ports to enumerate.

Triggers a pulse on a single bit for a specified duration.

```
function pulse( bitno: LongInt, ms_duration: LongInt): LongInt
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno the bit number; lowest bit has index 0

ms_duration desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→registerValueCallback()
digitalio.registerValueCallback()

YDigitalIO

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYDigitalIOValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

digitalio→**set_bitDirection()****YDigitalIO****digitalio**→**setBitDirection()****digitalio.set_bitDirection()**

Changes the direction of a single bit from the I/O port.

```
function set_bitDirection( bitno: LongInt, bitdirection: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0

bitdirection direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_bitOpenDrain()**
digitalio→**setBitOpenDrain()**
digitalio.set_bitOpenDrain()

YDigitalIO

Changes the electrical interface of a single bit from the I/O port.

```
function set_bitOpenDrain( bitno: LongInt, opendrain: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0
opendrain 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output.
Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_bitPolarity()****YDigitalIO****digitalio**→**setBitPolarity()****digitalio.set_bitPolarity()**

Changes the polarity of a single bit from the I/O port.

```
function set_bitPolarity( bitno: LongInt, bitpolarity: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0.

bitpolarity polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_bitState()****YDigitalIO****digitalio**→**setBitState()****digitalio.set_bitState()**

Sets a single bit of the I/O port.

```
function set_bitState( bitno: LongInt, bitstate: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0

bitstate the state of the bit (1 or 0)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_logicalName()****YDigitalIO****digitalio**→**setLogicalName()****digitalio.set_logicalName()**

Changes the logical name of the digital IO port.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the digital IO port.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

digitalio→**set_outputVoltage()**
digitalio→**setOutputVoltage()**
digitalio.set_outputVoltage()

YDigitalIO

Changes the voltage source used to drive output bits.

```
function set_outputVoltage( newval: Integer): integer
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portDirection()****YDigitalIO****digitalio**→**setPortDirection()****digitalio.set_portDirection()**

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

```
function set_portDirection( newval: LongInt): integer
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portOpenDrain()**
digitalio→**setPortOpenDrain()**
digitalio.set_portOpenDrain()

YDigitalIO

Changes the electrical interface for each bit of the port.

```
function set_portOpenDrain( newval: LongInt): integer
```

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the electrical interface for each bit of the port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portPolarity()**

YDigitalIO

digitalio→**setPortPolarity()****digitalio.set_portPolarity()**

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

```
function set_portPolarity( newval: LongInt): integer
```

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

Parameters :

newval an integer corresponding to the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_portState()****YDigitalIO****digitalio**→**setPortState()****digitalio.set_portState()**

Changes the digital IO port state: bit 0 represents input 0, and so on.

```
function set_portState( newval: LongInt): integer
```

This function has no effect on bits configured as input in `portDirection`.

Parameters :

newval an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

digitalio→**set_userData()**

YDigitalIO

digitalio→**setUserData()****digitalio.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

digitalio→**toggle_bitState()****digitalio.toggle_bitState()****YDigitalIO**

Reverts a single bit of the I/O port.

```
function toggle_bitState( bitno: LongInt): LongInt
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.13. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_display.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDisplay = yoctolib.YDisplay;
php	require_once('yocto_display.php');
c++	#include "yocto_display.h"
m	#import "yocto_display.h"
pas	uses yocto_display;
vb	yocto_display.vb
cs	yocto_display.cs
java	import com.yoctopuce.YoctoAPI.YDisplay;
py	from yocto_display import *

Global functions

yFindDisplay(func)

Retrieves a display for a given identifier.

yFirstDisplay()

Starts the enumeration of displays currently accessible.

YDisplay methods

display→copyLayerContent(srcLayerId, dstLayerId)

Copies the whole content of a layer to another layer.

display→describe()

Returns a short text that describes unambiguously the instance of the display in the form
TYPE (NAME) =SERIAL . FUNCTIONID.

display→fade(brightness, duration)

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

display→get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

display→get_brightness()

Returns the luminosity of the module informative leds (from 0 to 100).

display→get_displayHeight()

Returns the display height, in pixels.

display→get_displayLayer(layerId)

Returns a YDisplayLayer object that can be used to draw on the specified layer.

display→get_displayType()

Returns the display type: monochrome, gray levels or full color.

display→get_displayWidth()

Returns the display width, in pixels.

display→get_enabled()

Returns true if the screen is powered, false otherwise.

display→get_errorMessage()

Returns the error message of the latest error with the display.

display→get_errorType()

Returns the numerical error code of the latest error with the display.

display→get_friendlyName()

Returns a global identifier of the display in the format `MODULE_NAME . FUNCTION_NAME`.

display→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

display→get_functionId()

Returns the hardware identifier of the display, without reference to the module.

display→get_hardwareId()

Returns the unique hardware identifier of the display in the form `SERIAL . FUNCTIONID`.

display→get_layerCount()

Returns the number of available layers to draw on.

display→get_layerHeight()

Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()

Returns the width of the layers to draw on, in pixels.

display→get_logicalName()

Returns the logical name of the display.

display→get_module()

Gets the `YModule` object for the device on which the function is located.

display→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

display→get_orientation()

Returns the currently selected display orientation.

display→get_startupSeq()

Returns the name of the sequence to play when the displayed is powered on.

display→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

display→isOnline()

Checks if the display is currently reachable, without raising any error.

display→isOnline_async(callback, context)

Checks if the display is currently reachable, without raising any error (asynchronous version).

display→load(msValidity)

Preloads the display cache with a specified validity duration.

display→load_async(msValidity, callback, context)

Preloads the display cache with a specified validity duration (asynchronous version).

display→newSequence()

Starts to record all display commands into a sequence, for later replay.

display→nextDisplay()

Continues the enumeration of displays started using `yFirstDisplay()`.

display→pauseSequence(delay_ms)

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

display→playSequence(sequenceName)

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

display→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

display→resetAll()

Clears the display screen and resets all display layers to their default state.

display→saveSequence(sequenceName)

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

display→set_brightness(newval)

Changes the brightness of the display.

display→set_enabled(newval)

Changes the power state of the display.

display→set_logicalName(newval)

Changes the logical name of the display.

display→set_orientation(newval)

Changes the display orientation.

display→set_startupSeq(newval)

Changes the name of the sequence to play when the displayed is powered on.

display→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

display→stopSequence()

Stops immediately any ongoing sequence replay.

display→swapLayerContent(layerIdA, layerIdB)

Swaps the whole content of two layers.

display→upload(pathname, content)

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

display→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDisplay.FindDisplay() yFindDisplay()yFindDisplay()

YDisplay

Retrieves a display for a given identifier.

```
function yFindDisplay( func: string): TYDisplay
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the display

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FirstDisplay() yFirstDisplay()yFirstDisplay()

YDisplay

Starts the enumeration of displays currently accessible.

```
function yFirstDisplay( ): TYDisplay
```

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

Returns :

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

display→copyLayerContent()
display.copyLayerContent()

YDisplay

Copies the whole content of a layer to another layer.

```
function copyLayerContent( srcLayerId: LongInt,  
                           dstLayerId: LongInt): LongInt
```

The color and transparency of all the pixels from the destination layer are set to match the source pixels. This method only affects the displayed content, but does not change any property of the layer object. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

srcLayerId the identifier of the source layer (a number in range 0..layerCount-1)

dstLayerId the identifier of the destination layer (a number in range 0..layerCount-1)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→describe()**display.describe()****YDisplay**

Returns a short text that describes unambiguously the instance of the display in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the display (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

display→fade()display.fade()**YDisplay**

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

```
function fade( brightness: LongInt, duration: LongInt): LongInt
```

Parameters :

brightness the new screen brightness

duration duration of the brightness transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**get_advertisedValue()**

YDisplay

display→**advertisedValue()**

display.get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the display (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

display→**get_brightness()****YDisplay****display**→**brightness()****display.get_brightness()**

Returns the luminosity of the module informative leds (from 0 to 100).

```
function get_brightness( ): LongInt
```

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y_BRIGHTNESS_INVALID.

display→**get_displayHeight()**

YDisplay

display→**displayHeight()****display.get_displayHeight()**

Returns the display height, in pixels.

```
function get_displayHeight( ): LongInt
```

Returns :

an integer corresponding to the display height, in pixels

On failure, throws an exception or returns `Y_DISPLAYHEIGHT_INVALID`.

display→**get_displayType()****YDisplay****display**→**displayType()****display.get_displayType()**

Returns the display type: monochrome, gray levels or full color.

function **get_displayType**(): Integer

Returns :

a value among Y_DISPLAYTYPE_MONO, Y_DISPLAYTYPE_GRAY and Y_DISPLAYTYPE_RGB corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns Y_DISPLAYTYPE_INVALID.

display→**get_displayWidth()**

YDisplay

display→**displayWidth()****display.get_displayWidth()**

Returns the display width, in pixels.

```
function get_displayWidth( ): LongInt
```

Returns :

an integer corresponding to the display width, in pixels

On failure, throws an exception or returns `Y_DISPLAYWIDTH_INVALID`.

display→**get_enabled()****YDisplay****display**→**enabled()****display.get_enabled()**

Returns true if the screen is powered, false otherwise.

```
function get_enabled( ): Integer
```

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

display→**get_errorMessage()**

YDisplay

display→**errorMessage()****display.get_errorMessage()**

Returns the error message of the latest error with the display.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the display object

display→**get_errorType()****YDisplay****display**→**errorType()****display.get_errorType()**

Returns the numerical error code of the latest error with the display.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the display object

display→get_functionDescriptor()
display→functionDescriptor()
display.get_functionDescriptor()

YDisplay

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

display→**get_layerCount()****YDisplay****display**→**layerCount()****display.get_layerCount()**

Returns the number of available layers to draw on.

```
function get_layerCount( ): LongInt
```

Returns :

an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns `Y_LAYERCOUNT_INVALID`.

display→**get_layerHeight()**

YDisplay

display→**layerHeight()****display.get_layerHeight()**

Returns the height of the layers to draw on, in pixels.

```
function get_layerHeight( ): LongInt
```

Returns :

an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns `Y_LAYERHEIGHT_INVALID`.

display→**get_layerWidth()****YDisplay****display**→**layerWidth()****display.get_layerWidth()**

Returns the width of the layers to draw on, in pixels.

```
function get_layerWidth( ): LongInt
```

Returns :

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns `Y_LAYERWIDTH_INVALID`.

display→**get_logicalName()**

YDisplay

display→**logicalName()****display.get_logicalName()**

Returns the logical name of the display.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the display. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

display→**get_module()****YDisplay****display**→**module()****display.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

display→**get_orientation()**

YDisplay

display→**orientation()****display.get_orientation()**

Returns the currently selected display orientation.

function **get_orientation**(): Integer

Returns :

a value among Y_ORIENTATION_LEFT, Y_ORIENTATION_UP, Y_ORIENTATION_RIGHT and Y_ORIENTATION_DOWN corresponding to the currently selected display orientation

On failure, throws an exception or returns Y_ORIENTATION_INVALID.

display→**get_startupSeq()****YDisplay****display**→**startupSeq()****display.get_startupSeq()**

Returns the name of the sequence to play when the displayed is powered on.

```
function get_startupSeq( ): string
```

Returns :

a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns `Y_STARTUPSEQ_INVALID`.

display→**get_userData()**

YDisplay

display→**userData()****display.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

display→isOnline()display.isOnline()**YDisplay**

Checks if the display is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

Returns :

`true` if the display can be reached, and `false` otherwise

display→load()display.load()**YDisplay**

Preloads the display cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

display→newSequence()display.newSequence()

YDisplay

Starts to record all display commands into a sequence, for later replay.

```
function newSequence( ): LongInt
```

The name used to store the sequence is specified when calling `saveSequence()`, once the recording is complete.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**nextDisplay()****display.nextDisplay()**

YDisplay

Continues the enumeration of displays started using `yFirstDisplay()`.

```
function nextDisplay( ): TYDisplay
```

Returns :

a pointer to a `YDisplay` object, corresponding to a display currently online, or a `null` pointer if there are no more displays to enumerate.

display→pauseSequence()display.pauseSequence()**YDisplay**

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

```
function pauseSequence( delay_ms: LongInt): LongInt
```

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

Parameters :

delay_ms the duration to wait, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→playSequence()display.playSequence()**YDisplay**

Replays a display sequence previously recorded using `newSequence ()` and `saveSequence ()`.

```
function playSequence( sequenceName: string): LongInt
```

Parameters :

sequenceName the name of the newly created sequence

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→registerValueCallback()
display.registerValueCallback()

YDisplay

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYDisplayValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

display→resetAll()display.resetAll()

YDisplay

Clears the display screen and resets all display layers to their default state.

```
function resetAll( ): LongInt
```

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→saveSequence()display.saveSequence()**YDisplay**

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```
function saveSequence( sequenceName: string): LongInt
```

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_brightness()****YDisplay****display**→**setBrightness()****display.set_brightness()**

Changes the brightness of the display.

```
function set_brightness( newval: LongInt): integer
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the brightness of the display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_enabled()****YDisplay****display**→**setEnabled()****display.set_enabled()**

Changes the power state of the display.

```
function set_enabled( newval: Integer): integer
```

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the power state of the display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_logicalName()**

YDisplay

display→**setLogicalName()****display.set_logicalName()**

Changes the logical name of the display.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the display.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

display→**set_orientation()****YDisplay****display**→**setOrientation()****display.set_orientation()**

Changes the display orientation.

```
function set_orientation( newval: Integer): integer
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the display orientation

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_startupSeq()**

YDisplay

display→**setStartupSeq()****display.set_startupSeq()**

Changes the name of the sequence to play when the displayed is powered on.

```
function set_startupSeq( newval: string): integer
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the name of the sequence to play when the displayed is powered on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→**set_userData()****YDisplay****display**→**setUserData()****display.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userData( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

display→stopSequence()display.stopSequence()**YDisplay**

Stops immediately any ongoing sequence replay.

```
function stopSequence( ): LongInt
```

The display is left as is.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→swapLayerContent()
display.swapLayerContent()

YDisplay

Swaps the whole content of two layers.

```
function swapLayerContent( layerIdA: LongInt, layerIdB: LongInt): LongInt
```

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between one hidden layer and a visible layer, this method makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

layerIdA the first layer (a number in range 0..layerCount-1)

layerIdB the second layer (a number in range 0..layerCount-1)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→upload()display.upload()**YDisplay**

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

```
function upload( pathname: string, content: TByteArray): LongInt
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.14. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_display.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDisplay = yoctolib.YDisplay;
php	require_once('yocto_display.php');
c++	#include "yocto_display.h"
m	#import "yocto_display.h"
pas	uses yocto_display;
vb	yocto_display.vb
cs	yocto_display.cs
java	import com.yoctopuce.YoctoAPI.YDisplay;
py	from yocto_display import *

YDisplayLayer methods

displaylayer→clear()

Erases the whole content of the layer (makes it fully transparent).

displaylayer→clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

displaylayer→consoleOut(text)

Outputs a message in the console area, and advances the console pointer accordingly.

displaylayer→drawBar(x1, y1, x2, y2)

Draws a filled rectangular bar at a specified position.

displaylayer→drawBitmap(x, y, w, bitmap, bgcolor)

Draws a bitmap at the specified position.

displaylayer→drawCircle(x, y, r)

Draws an empty circle at a specified position.

displaylayer→drawDisc(x, y, r)

Draws a filled disc at a given position.

displaylayer→drawImage(x, y, imagename)

Draws a GIF image at the specified position.

displaylayer→drawPixel(x, y)

Draws a single pixel at the specified position.

displaylayer→drawRect(x1, y1, x2, y2)

Draws an empty rectangle at a specified position.

displaylayer→drawText(x, y, anchor, text)

Draws a text string at the specified position.

displaylayer→get_display()

Gets parent YDisplay.

displaylayer→get_displayHeight()

Returns the display height, in pixels.

displaylayer→get_displayWidth()

Returns the display width, in pixels.

displaylayer→get_layerHeight()

Returns the height of the layers to draw on, in pixels.

displaylayer→get_layerWidth()

Returns the width of the layers to draw on, in pixels.

displaylayer→hide()

Hides the layer.

displaylayer→lineTo(x, y)

Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(x, y)

Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()

Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(color)

Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

displaylayer→selectFont(fontname)

Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(graylevel)

Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(mode)

Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(bgcol)

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(x1, y1, x2, y2)

Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(wordwrap)

Sets up the wrapping behaviour used by the `consoleOut` function.

displaylayer→setLayerPosition(x, y, scrollTime)

Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()

Shows the layer.

displaylayer→clear()displaylayer.clear()**YDisplayLayer**

Erases the whole content of the layer (makes it fully transparent).

```
function clear( ): LongInt
```

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset ()` instead.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→clearConsole()
displaylayer.clearConsole()

YDisplayLayer

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

```
function clearConsole( ): LongInt
```

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→consoleOut()displaylayer.consoleOut()

YDisplayLayer

Outputs a message in the console area, and advances the console pointer accordingly.

```
function consoleOut( text: string): LongInt
```

The console pointer position is automatically moved to the beginning of the next line when a newline character is met, or when the right margin is hit. When the new text to display extends below the lower margin, the console area is automatically scrolled up.

Parameters :

text the message to display

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBar()displaylayer.drawBar()**YDisplayLayer**

Draws a filled rectangular bar at a specified position.

```
function drawBar( x1: LongInt,  
                  y1: LongInt,  
                  x2: LongInt,  
                  y2: LongInt): LongInt
```

Parameters :

- x1** the distance from left of layer to the left border of the rectangle, in pixels
- y1** the distance from top of layer to the top border of the rectangle, in pixels
- x2** the distance from left of layer to the right border of the rectangle, in pixels
- y2** the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawBitmap()
displaylayer.drawBitmap()****YDisplayLayer**

Draws a bitmap at the specified position.

```
function drawBitmap( x: LongInt,  
                    y: LongInt,  
                    w: LongInt,  
                    bitmap: TByteArray,  
                    bgcol: LongInt): LongInt
```

The bitmap is provided as a binary object, where each pixel maps to a bit, from left to right and from top to bottom. The most significant bit of each byte maps to the leftmost pixel, and the least significant bit maps to the rightmost pixel. Bits set to 1 are drawn using the layer selected pen color. Bits set to 0 are drawn using the specified background gray level, unless -1 is specified, in which case they are not drawn at all (as if transparent).

Parameters :

- x** the distance from left of layer to the left of the bitmap, in pixels
- y** the distance from top of layer to the top of the bitmap, in pixels
- w** the width of the bitmap, in pixels
- bitmap** a binary object
- bgcol** the background gray level to use for zero bits (0 = black, 255 = white), or -1 to leave the pixels unchanged

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawCircle()displaylayer.drawCircle()**YDisplayLayer**

Draws an empty circle at a specified position.

```
function drawCircle( x: LongInt, y: LongInt, r: LongInt): LongInt
```

Parameters :

- x** the distance from left of layer to the center of the circle, in pixels
- y** the distance from top of layer to the center of the circle, in pixels
- r** the radius of the circle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawDisc()displaylayer.drawDisc()**YDisplayLayer**

Draws a filled disc at a given position.

```
function drawDisc( x: LongInt, y: LongInt, r: LongInt): LongInt
```

Parameters :

- x** the distance from left of layer to the center of the disc, in pixels
- y** the distance from top of layer to the center of the disc, in pixels
- r** the radius of the disc, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawImage()displaylayer.drawImage()**YDisplayLayer**

Draws a GIF image at the specified position.

```
function drawImage( x: LongInt, y: LongInt, imagename: string): LongInt
```

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

Parameters :

x the distance from left of layer to the left of the image, in pixels
y the distance from top of layer to the top of the image, in pixels
imagename the GIF file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawPixel()displaylayer.drawPixel()

YDisplayLayer

Draws a single pixel at the specified position.

```
function drawPixel( x: LongInt, y: LongInt): LongInt
```

Parameters :

- x** the distance from left of layer, in pixels
- y** the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawRect()displaylayer.drawRect()**YDisplayLayer**

Draws an empty rectangle at a specified position.

```
function drawRect( x1: LongInt,  
                   y1: LongInt,  
                   x2: LongInt,  
                   y2: LongInt): LongInt
```

Parameters :

- x1** the distance from left of layer to the left border of the rectangle, in pixels
- y1** the distance from top of layer to the top border of the rectangle, in pixels
- x2** the distance from left of layer to the right border of the rectangle, in pixels
- y2** the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawText(displaylayer.drawText())**YDisplayLayer**

Draws a text string at the specified position.

```
function drawText( x: LongInt,
                  y: LongInt,
                  anchor: TYALIGN,
                  text: string): LongInt
```

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

Parameters :

- x** the distance from left of layer to the text anchor point, in pixels
- y** the distance from top of layer to the text anchor point, in pixels
- anchor** the text anchor point, chosen among the Y_ALIGN enumeration: Y_ALIGN_TOP_LEFT, Y_ALIGN_CENTER_LEFT, Y_ALIGN_BASELINE_LEFT, Y_ALIGN_BOTTOM_LEFT, Y_ALIGN_TOP_CENTER, Y_ALIGN_CENTER, Y_ALIGN_BASELINE_CENTER, Y_ALIGN_BOTTOM_CENTER, Y_ALIGN_TOP_DECIMAL, Y_ALIGN_CENTER_DECIMAL, Y_ALIGN_BASELINE_DECIMAL, Y_ALIGN_BOTTOM_DECIMAL, Y_ALIGN_TOP_RIGHT, Y_ALIGN_CENTER_RIGHT, Y_ALIGN_BASELINE_RIGHT, Y_ALIGN_BOTTOM_RIGHT.
- text** the text string to draw

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→**get_display()**

YDisplayLayer

displaylayer→**display()****displaylayer.get_display()**

Gets parent YDisplay.

```
function get_display( ): TYDisplay
```

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :

an YDisplay object

displaylayer→get_displayHeight()
displaylayer→displayHeight()
displaylayer.get_displayHeight()

YDisplayLayer

Returns the display height, in pixels.

```
function get_displayHeight( ): LongInt
```

Returns :

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth()

YDisplayLayer

displaylayer→displayWidth()

displaylayer.get_displayWidth()

Returns the display width, in pixels.

```
function get_displayWidth( ): LongInt
```

Returns :

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→get_layerHeight()
displaylayer→layerHeight()
displaylayer.get_layerHeight()

YDisplayLayer

Returns the height of the layers to draw on, in pixels.

```
function get_layerHeight( ): LongInt
```

Returns :

an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth()

YDisplayLayer

displaylayer→layerWidth()

displaylayer.get_layerWidth()

Returns the width of the layers to draw on, in pixels.

function **get_layerWidth**(): LongInt

Returns :

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERWIDTH_INVALID.

displaylayer→hide()displaylayer.hide()

YDisplayLayer

Hides the layer.

```
function hide( ): LongInt
```

The state of the layer is perserved but the layer is not displayed on the screen until the next call to `unhide()`. Hiding the layer can positively affect the drawing speed, since it postpones the rendering until all operations are completed (double-buffering).

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→lineTo()displaylayer.lineTo()**YDisplayLayer**

Draws a line from current drawing pointer position to the specified position.

```
function lineTo( x: LongInt, y: LongInt): LongInt
```

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

Parameters :

- x** the distance from left of layer to the end point of the line, in pixels
- y** the distance from top of layer to the end point of the line, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→moveTo()displaylayer.moveTo()**YDisplayLayer**

Moves the drawing pointer of this layer to the specified position.

```
function moveTo( x: LongInt, y: LongInt): LongInt
```

Parameters :

- x** the distance from left of layer, in pixels
- y** the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→reset()displaylayer.reset()**YDisplayLayer**

Reverts the layer to its initial state (fully transparent, default settings).

```
function reset( ): LongInt
```

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectColorPen()
displaylayer.selectColorPen()**

YDisplayLayer

Selects the pen color for all subsequent drawing functions, including text drawing.

```
function selectColorPen( color: LongInt): LongInt
```

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectEraser()
displaylayer.selectEraser()****YDisplayLayer**

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

```
function selectEraser( ): LongInt
```

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectFont()displaylayer.selectFont()**YDisplayLayer**

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```
function selectFont( fontname: string): LongInt
```

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

Parameters :

fontname the font file name

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectGrayPen()
displaylayer.selectGrayPen()****YDisplayLayer**

Selects the pen gray level for all subsequent drawing functions, including text drawing.

```
function selectGrayPen( graylevel: LongInt): LongInt
```

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the highest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

Parameters :

graylevel the desired gray level, from 0 to 255

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setAntialiasingMode()
displaylayer.setAntialiasingMode()

YDisplayLayer

Enables or disables anti-aliasing for drawing oblique lines and circles.

```
function setAntialiasingMode( mode: boolean): LongInt
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

Parameters :

mode true to enable antialiasing, false to disable it.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleBackground() displaylayer.setConsoleBackground()

YDisplayLayer

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

```
function setConsoleBackground( bgcol: LongInt): LongInt
```

Parameters :

bgcol the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleMargins()
displaylayer.setConsoleMargins()

YDisplayLayer

Sets up display margins for the `consoleOut` function.

```
function setConsoleMargins( x1: LongInt,  
                           y1: LongInt,  
                           x2: LongInt,  
                           y2: LongInt): LongInt
```

Parameters :

- x1** the distance from left of layer to the left margin, in pixels
- y1** the distance from top of layer to the top margin, in pixels
- x2** the distance from left of layer to the right margin, in pixels
- y2** the distance from top of layer to the bottom margin, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleWordWrap()
displaylayer.setConsoleWordWrap()**YDisplayLayer**

Sets up the wrapping behaviour used by the `consoleOut` function.

```
function setConsoleWordWrap( wordwrap: boolean): LongInt
```

Parameters :

wordwrap `true` to wrap only between words, `false` to wrap on the last column anyway.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setLayerPosition()
displaylayer.setLayerPosition()**YDisplayLayer**

Sets the position of the layer relative to the display upper left corner.

```
function setLayerPosition( x: LongInt,  
                           y: LongInt,  
                           scrollTime: LongInt): LongInt
```

When smooth scrolling is used, the display offset of the layer is automatically updated during the next milliseconds to animate the move of the layer.

Parameters :

- x** the distance from left of display to the upper left corner of the layer
- y** the distance from top of display to the upper left corner of the layer
- scrollTime** number of milliseconds to use for smooth scrolling, or 0 if the scrolling should be immediate.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→unhide()displaylayer.unhide()

YDisplayLayer

Shows the layer.

```
function unhide( ): LongInt
```

Shows the layer again after a hide command.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.15. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_dualpower.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YDualPower = yoctolib.YDualPower;
php	require_once('yocto_dualpower.php');
c++	#include "yocto_dualpower.h"
m	#import "yocto_dualpower.h"
pas	uses yocto_dualpower;
vb	yocto_dualpower.vb
cs	yocto_dualpower.cs
java	import com.yoctopuce.YoctoAPI.YDualPower;
py	from yocto_dualpower import *

Global functions

yFindDualPower(func)

Retrieves a dual power control for a given identifier.

yFirstDualPower()

Starts the enumeration of dual power controls currently accessible.

YDualPower methods

dualpower→describe()

Returns a short text that describes unambiguously the instance of the power control in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

dualpower→get_advertisedValue()

Returns the current value of the power control (no more than 6 characters).

dualpower→get_errorMessage()

Returns the error message of the latest error with the power control.

dualpower→get_errorType()

Returns the numerical error code of the latest error with the power control.

dualpower→get_extVoltage()

Returns the measured voltage on the external power source, in millivolts.

dualpower→get_friendlyName()

Returns a global identifier of the power control in the format `MODULE_NAME . FUNCTION_NAME`.

dualpower→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

dualpower→get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

dualpower→get_hardwareId()

Returns the unique hardware identifier of the power control in the form `SERIAL . FUNCTIONID`.

dualpower→get_logicalName()

Returns the logical name of the power control.

dualpower→get_module()

	Gets the <code>YModule</code> object for the device on which the function is located.
<code>dualpower→get_module_async(callback, context)</code>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<code>dualpower→get_powerControl()</code>	Returns the selected power source for module functions that require lots of current.
<code>dualpower→get_powerState()</code>	Returns the current power source for module functions that require lots of current.
<code>dualpower→get_userData()</code>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<code>dualpower→isOnline()</code>	Checks if the power control is currently reachable, without raising any error.
<code>dualpower→isOnline_async(callback, context)</code>	Checks if the power control is currently reachable, without raising any error (asynchronous version).
<code>dualpower→load(msValidity)</code>	Preloads the power control cache with a specified validity duration.
<code>dualpower→load_async(msValidity, callback, context)</code>	Preloads the power control cache with a specified validity duration (asynchronous version).
<code>dualpower→nextDualPower()</code>	Continues the enumeration of dual power controls started using <code>yFirstDualPower()</code> .
<code>dualpower→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>dualpower→set_logicalName(newval)</code>	Changes the logical name of the power control.
<code>dualpower→set_powerControl(newval)</code>	Changes the selected power source for module functions that require lots of current.
<code>dualpower→set_userData(data)</code>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<code>dualpower→wait_async(callback, context)</code>	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDualPower.FindDualPower() yFindDualPower()yFindDualPower()

YDualPower

Retrieves a dual power control for a given identifier.

```
function yFindDualPower( func: string): TYDualPower
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDualPower.isOnline()` to test if the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

YDualPower.FirstDualPower() yFirstDualPower()yFirstDualPower()

YDualPower

Starts the enumeration of dual power controls currently accessible.

```
function yFirstDualPower( ): TYDualPower
```

Use the method `YDualPower.nextDualPower()` to iterate on next dual power controls.

Returns :

a pointer to a `YDualPower` object, corresponding to the first dual power control currently online, or a `null` pointer if there are none.

dualpower→describe()**dualpower.describe()****YDualPower**

Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the power control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

dualpower→get_advertisedValue()

YDualPower

dualpower→advertisedValue()

dualpower.get_advertisedValue()

Returns the current value of the power control (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the power control (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

dualpower→get_errorMessage()**YDualPower****dualpower→errorMessage()****dualpower.get_errorMessage()**

Returns the error message of the latest error with the power control.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the power control object

dualpower→**get_errorType()**

YDualPower

dualpower→**errorType()****dualpower.get_errorType()**

Returns the numerical error code of the latest error with the power control.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the power control object

dualpower→get_extVoltage()**YDualPower****dualpower→extVoltage()dualpower.get_extVoltage()**

Returns the measured voltage on the external power source, in millivolts.

function **get_extVoltage()**: LongInt

Returns :

an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns Y_EXTVOLTAGE_INVALID.

dualpower→**get_functionDescriptor()**
dualpower→**functionDescriptor()**
dualpower.get_functionDescriptor()

YDualPower

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

dualpower→get_logicalName()**YDualPower****dualpower→logicalName()****dualpower.get_logicalName()**

Returns the logical name of the power control.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the power control. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

dualpower→**get_module()**

YDualPower

dualpower→**module()****dualpower.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module()**: YModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

dualpower→**get_powerControl()****YDualPower****dualpower**→**powerControl()****dualpower.get_powerControl()**

Returns the selected power source for module functions that require lots of current.

function **get_powerControl**(): Integer

Returns :

a value among Y_POWERCONTROL_AUTO, Y_POWERCONTROL_FROM_USB, Y_POWERCONTROL_FROM_EXT and Y_POWERCONTROL_OFF corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERCONTROL_INVALID.

dualpower→get_powerState()

YDualPower

dualpower→powerState()

dualpower.get_powerState()

Returns the current power source for module functions that require lots of current.

function **get_powerState**(): Integer

Returns :

a value among Y_POWERSTATE_OFF, Y_POWERSTATE_FROM_USB and Y_POWERSTATE_FROM_EXT corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns Y_POWERSTATE_INVALID.

dualpower→**get_userdata()****YDualPower****dualpower**→**userData()****dualpower.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

function **get_userdata()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

dualpower→**isOnline()****dualpower.isOnline()**

YDualPower

Checks if the power control is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power control.

Returns :

`true` if the power control can be reached, and `false` otherwise

dualpower→load()dualpower.load()**YDualPower**

Preloads the power control cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

dualpower→**nextDualPower()**
dualpower.nextDualPower()

YDualPower

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

```
function nextDualPower( ): TYDualPower
```

Returns :

a pointer to a `YDualPower` object, corresponding to a dual power control currently online, or a `null` pointer if there are no more dual power controls to enumerate.

dualpower→registerValueCallback()
dualpower.registerValueCallback()

YDualPower

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYDualPowerValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

dualpower→**set_logicalName()****YDualPower****dualpower**→**setLogicalName()****dualpower.set_logicalName()**

Changes the logical name of the power control.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the power control.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

dualpower→set_powerControl()**YDualPower****dualpower→setPowerControl()****dualpower.set_powerControl()**

Changes the selected power source for module functions that require lots of current.

```
function set_powerControl( newval: Integer): integer
```

Parameters :

newval a value among Y_POWERCONTROL_AUTO, Y_POWERCONTROL_FROM_USB, Y_POWERCONTROL_FROM_EXT and Y_POWERCONTROL_OFF corresponding to the selected power source for module functions that require lots of current

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

dualpower→**set_userdata()**

YDualPower

dualpower→**setUserData()****dualpower.set_userdata()**

Stores a user context provided as argument in the `userData` attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.16. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_files.js'></script>
nodejs	var yocotolib = require('yocotolib'); var YFiles = yocotolib.YFiles;
php	require_once('yocto_files.php');
c++	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *

Global functions

yFindFiles(func)

Retrieves a filesystem for a given identifier.

yFirstFiles()

Starts the enumeration of filesystems currently accessible.

YFiles methods

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form TYPE (NAME) = SERIAL . FUNCTIONID.

files→download(pathname)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(pathname, callback, context)

Downloads the requested file and returns a binary buffer with its content.

files→format_fs()

Reinitializes the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

Returns a global identifier of the filesystem in the format MODULE_NAME . FUNCTION_NAME.

files→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

files→get_functionId()

	Returns the hardware identifier of the filesystem, without reference to the module.
files→get_hardwareId()	Returns the unique hardware identifier of the filesystem in the form <code>SERIAL.FUNCTIONID</code> .
files→get_list(pattern)	Returns a list of <code>YFileRecord</code> objects that describe files currently loaded in the filesystem.
files→get_logicalName()	Returns the logical name of the filesystem.
files→get_module()	Gets the <code>YModule</code> object for the device on which the function is located.
files→get_module_async(callback, context)	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
files→get_userData()	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
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 <code>yFirstFiles()</code> .
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 <code>userData</code> attribute of the function.
files→upload(pathname, content)	Uploads a file to the filesystem, to the specified full path name.
files→wait_async(callback, context)	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YFiles.FindFiles() yFindFiles()yFindFiles()

YFiles

Retrieves a filesystem for a given identifier.

```
function yFindFiles( func: string): TYFiles
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the filesystem

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles()**YFiles****yFirstFiles()**

Starts the enumeration of filesystems currently accessible.

```
function yFirstFiles( ): TYFiles
```

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

files→describe()files.describe()**YFiles**

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the filesystem (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

files→download(files.download())**YFiles**

Downloads the requested file and returns a binary buffer with its content.

```
function download( pathname: string): TByteArray
```

Parameters :

pathname path and name of the file to download

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

files→format_fs()files.format_fs()

YFiles

Reinitializes the filesystem to its clean, unfragmented, empty state.

```
function format_fs( ): LongInt
```

All files previously uploaded are permanently lost.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→**get_advertisedValue()**

YFiles

files→**advertisedValue()****files.get_advertisedValue()**

Returns the current value of the filesystem (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the filesystem (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

files→**get_errorMessage()****YFiles****files**→**errorMessage()****files.get_errorMessage()**

Returns the error message of the latest error with the filesystem.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the filesystem object

files→**get_errorType()**

YFiles

files→**errorType()****files.get_errorType()**

Returns the numerical error code of the latest error with the filesystem.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the filesystem object

files→**get_filesCount()****YFiles****files**→**filesCount()****files.get_filesCount()**

Returns the number of files currently loaded in the filesystem.

function **get_filesCount**(): LongInt

Returns :

an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `Y_FILESCOUNT_INVALID`.

files→**get_freeSpace()**

YFiles

files→**freeSpace()****files.get_freeSpace()**

Returns the free space for uploading new files to the filesystem, in bytes.

function **get_freeSpace**(): LongInt

Returns :

an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns `Y_FREESPACE_INVALID`.

files→**get_functionDescriptor()****YFiles****files**→**functionDescriptor()****files.get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

files→**get_list()****YFiles****files**→**list()****files.get_list()**

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
function get_list( pattern: string): TYFileRecordArray
```

Parameters :

pattern an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

Returns :

a list of YFileRecord objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→**get_logicalName()****YFiles****files**→**logicalName()****files.get_logicalName()**

Returns the logical name of the filesystem.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the filesystem. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

files→**get_module()**

YFiles

files→**module()****files.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module**(): TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

files→**get_userData()****YFiles****files**→**userData()****files.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

files→isOnline()**files.isOnline()****YFiles**

Checks if the filesystem is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

Returns :

`true` if the filesystem can be reached, and `false` otherwise

files→load()files.load()**YFiles**

Preloads the filesystem cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

files→**nextFiles()****files.nextFiles()**

YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

```
function nextFiles( ): TYFiles
```

Returns :

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

files→registerValueCallback()
files.registerValueCallback()

YFiles

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYFilesValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

files→remove()files.remove()**YFiles**

Deletes a file, given by its full path name, from the filesystem.

```
function remove( pathname: string): LongInt
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→**set_logicalName()****YFiles****files**→**setLogicalName()****files.set_logicalName()**

Changes the logical name of the filesystem.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the filesystem.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

files→**set_userdata()**

YFiles

files→**setUserData()****files.set_userdata()**

Stores a user context provided as argument in the `userData` attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

files→upload()files.upload()

YFiles

Uploads a file to the filesystem, to the specified full path name.

```
function upload( pathname: string, content: TByteArray): LongInt
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.17. GenericSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_genericsensor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YGenericSensor = yoctolib.YGenericSensor;
php	require_once('yocto_genericsensor.php');
c++	#include "yocto_genericsensor.h"
m	#import "yocto_genericsensor.h"
pas	uses yocto_genericsensor;
vb	yocto_genericsensor.vb
cs	yocto_genericsensor.cs
java	import com.yoctopuce.YoctoAPI.YGenericSensor;
py	from yocto_genericsensor import *

Global functions

yFindGenericSensor(func)

Retrieves a generic sensor for a given identifier.

yFirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

YGenericSensor methods

genericsensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

genericsensor→describe()

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

genericsensor→get_advertisedValue()

Returns the current value of the generic sensor (no more than 6 characters).

genericsensor→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

genericsensor→get_currentValue()

Returns the current measured value.

genericsensor→get_errorMessage()

Returns the error message of the latest error with the generic sensor.

genericsensor→get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

genericsensor→get_friendlyName()

Returns a global identifier of the generic sensor in the format `MODULE_NAME . FUNCTION_NAME`.

genericsensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

genericsensor→get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

genericsensor→get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form `SERIAL . FUNCTIONID`.

genericsensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

genericsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

genericsensor→get_logicalName()

Returns the logical name of the generic sensor.

genericsensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

genericsensor→get_module()

Gets the YModule object for the device on which the function is located.

genericsensor→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

genericsensor→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

genericsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

genericsensor→get_resolution()

Returns the resolution of the measured values.

genericsensor→get_signalRange()

Returns the electric signal range used by the sensor.

genericsensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericsensor→get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

genericsensor→get_unit()

Returns the measuring unit for the measure.

genericsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

genericsensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericsensor→isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

genericsensor→isOnline_async(callback, context)

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

genericsensor→load(msValidity)

Preloads the generic sensor cache with a specified validity duration.

genericsensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

genericsensor→load_async(msValidity, callback, context)

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

genericsensor→nextGenericSensor()

Continues the enumeration of generic sensors started using yFirstGenericSensor().

genericsensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

genericsensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

genericsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

genericsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

genericsensor→set_logicalName(newval)

Changes the logical name of the generic sensor.

genericsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

genericsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

genericsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

genericsensor→set_signalRange(newval)

Changes the electric signal range used by the sensor.

genericsensor→set_unit(newval)

Changes the measuring unit for the measured value.

genericsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

genericsensor→set_valueRange(newval)

Changes the physical value range measured by the sensor.

genericsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGenericSensor.FindGenericSensor() yFindGenericSensor()yFindGenericSensor()

YGenericSensor

Retrieves a generic sensor for a given identifier.

```
function yFindGenericSensor( func: string): TYGenericSensor
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the generic sensor

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FirstGenericSensor() yFirstGenericSensor()yFirstGenericSensor()

YGenericSensor

Starts the enumeration of generic sensors currently accessible.

```
function yFirstGenericSensor( ): TYGenericSensor
```

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

Returns :

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a `null` pointer if there are none.

genericsensor→calibrateFromPoints()
genericsensor.calibrateFromPoints()

YGenericSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→describe()genericsensor.describe()**YGenericSensor**

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the generic sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

genericsensor→get_advertisedValue()**YGenericSensor****genericsensor→advertisedValue()****genericsensor.get_advertisedValue()**

Returns the current value of the generic sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the generic sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

genericsensor→get_currentRawValue()

YGenericSensor

genericsensor→currentRawValue()

genericsensor.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

genericsensor→get_currentValue()
genericsensor→currentValue()
genericsensor.get_currentValue()

YGenericSensor

Returns the current measured value.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

genericsensor→get_errorMessage()
genericsensor→errorMessage()
genericsensor.get_errorMessage()

YGenericSensor

Returns the error message of the latest error with the generic sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the generic sensor object

genericsensor→get_errorType()
genericsensor→errorType()
genericsensor.get_errorType()

YGenericSensor

Returns the numerical error code of the latest error with the generic sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the generic sensor object

genericsensor→get_functionDescriptor()
genericsensor→functionDescriptor()
genericsensor.get_functionDescriptor()

YGenericSensor

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

genericsensor→**get_highestValue()****YGenericSensor****genericsensor**→**highestValue()****genericsensor.get_highestValue()**

Returns the maximal value observed for the measure since the device was started.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

genericsensor→get_logFrequency()
genericsensor→logFrequency()
genericsensor.get_logFrequency()

YGenericSensor

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

genericsensor→get_logicalName()**YGenericSensor****genericsensor→logicalName()****genericsensor.get_logicalName()**

Returns the logical name of the generic sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the generic sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

genericsensor→get_lowestValue()
genericsensor→lowestValue()
genericsensor.get_lowestValue()

YGenericSensor

Returns the minimal value observed for the measure since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

genericsensor→get_module()
genericsensor→module()
genericsensor.get_module()

YGenericSensor

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

genericsensor→**get_recordedData()**
genericsensor→**recordedData()**
genericsensor.get_recordedData()

YGenericSensor

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

genericsensor→get_reportFrequency()**YGenericSensor****genericsensor→reportFrequency()****genericsensor.get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

genericsensor→get_resolution()

YGenericSensor

genericsensor→resolution()

genericsensor.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

genericsensor→get_signalRange()**YGenericSensor****genericsensor→signalRange()****genericsensor.get_signalRange()**

Returns the electric signal range used by the sensor.

```
function get_signalRange( ): string
```

Returns :

a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns Y_SIGNALRANGE_INVALID.

genericsensor→get_signalUnit()

YGenericSensor

genericsensor→signalUnit()

genericsensor.get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

```
function get_signalUnit( ): string
```

Returns :

a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALUNIT_INVALID.

genericsensor→get_signalValue()**YGenericSensor****genericsensor→signalValue()****genericsensor.get_signalValue()**

Returns the measured value of the electrical signal used by the sensor.

```
function get_signalValue( ): double
```

Returns :

a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALVALUE_INVALID.

genericsensor→**get_unit()**

YGenericSensor

genericsensor→**unit()****genericsensor.get_unit()**

Returns the measuring unit for the measure.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

genericsensor→get_userData()**YGenericSensor****genericsensor→userData()****genericsensor.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

genericsensor→get_valueRange()
genericsensor→valueRange()
genericsensor.get_valueRange()

YGenericSensor

Returns the physical value range measured by the sensor.

```
function get_valueRange( ): string
```

Returns :

a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns Y_VALUERANGE_INVALID.

genericsensor→**isOnline()****genericsensor.isOnline()****YGenericSensor**

Checks if the generic sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the generic sensor.

Returns :

`true` if the generic sensor can be reached, and `false` otherwise

genericsensor→load()**genericsensor.load()****YGenericSensor**

Preloads the generic sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**genericsensor→loadCalibrationPoints()
genericsensor.loadCalibrationPoints()**

YGenericSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→nextGenericSensor()
genericsensor.nextGenericSensor()

YGenericSensor

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

function **nextGenericSensor**(): TYGenericSensor

Returns :

a pointer to a `YGenericSensor` object, corresponding to a generic sensor currently online, or a `null` pointer if there are no more generic sensors to enumerate.

**genericsensor→registerTimedReportCallback()
genericsensor.registerTimedReportCallback()**

YGenericSensor

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYGenericSensorTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

genericsensor→registerValueCallback()
genericsensor.registerValueCallback()**YGenericSensor**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYGenericSensorValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

genericsensor→set_highestValue()
genericsensor→setHighestValue()
genericsensor.set_highestValue()

YGenericSensor

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logFrequency()
genericsensor→setLogFrequency()
genericsensor.set_logFrequency()

YGenericSensor

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logicalName()**YGenericSensor****genericsensor→setLogicalName()****genericsensor.set_logicalName()**

Changes the logical name of the generic sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the generic sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

genericsensor→**set_lowestValue()**
genericsensor→**setLowestValue()**
genericsensor.set_lowestValue()

YGenericSensor

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_reportFrequency()
genericsensor→setReportFrequency()
genericsensor.set_reportFrequency()

YGenericSensor

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_resolution()
genericsensor→setResolution()
genericsensor.set_resolution()

YGenericSensor

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalRange()**YGenericSensor****genericsensor→setSignalRange()****genericsensor.set_signalRange()**

Changes the electric signal range used by the sensor.

```
function set_signalRange( newval: string): integer
```

Parameters :

newval a string corresponding to the electric signal range used by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→**set_unit()**

YGenericSensor

genericsensor→**setUnit()****genericsensor.set_unit()**

Changes the measuring unit for the measured value.

```
function set_unit( newval: string): integer
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the measuring unit for the measured value

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_userData()
genericsensor→setUserData()
genericsensor.set_userData()

YGenericSensor

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userData( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

genericsensor→set_valueRange()
genericsensor→setValueRange()
genericsensor.set_valueRange()

YGenericSensor

Changes the physical value range measured by the sensor.

```
function set_valueRange( newval: string): integer
```

The range change may have a side effect on the display resolution, as it may be adapted automatically.

Parameters :

newval a string corresponding to the physical value range measured by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.18. Gyroscope function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YGyro = yoctolib.YGyro;
php	require_once('yocto_gyro.php');
c++	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

Global functions

yFindGyro(func)

Retrieves a gyroscope for a given identifier.

yFirstGyro()

Starts the enumeration of gyroscopes currently accessible.

YGyro methods

gyro→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

gyro→describe()

Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

gyro→get_advertisedValue()

Returns the current value of the gyroscope (no more than 6 characters).

gyro→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

gyro→get_currentValue()

Returns the current value of the angular velocity.

gyro→get_errorMessage()

Returns the error message of the latest error with the gyroscope.

gyro→get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

gyro→get_friendlyName()

Returns a global identifier of the gyroscope in the format `MODULE_NAME . FUNCTION_NAME`.

gyro→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

gyro→get_functionId()

Returns the hardware identifier of the gyroscope, without reference to the module.

gyro→get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form `SERIAL . FUNCTIONID`.

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

gyro→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

gyro→get_logicalName()

Returns the logical name of the gyroscope.

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

gyro→get_module()

Gets the YModule object for the device on which the function is located.

gyro→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionW()

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionX()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionY()

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionZ()

Returns the z component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

gyro→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

gyro→get_resolution()

Returns the resolution of the measured values.

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_unit()

Returns the measuring unit for the angular velocity.

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

gyro→get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

gyro→get_yValue()

Returns the angular velocity around the Y axis of the device, as a floating point number.

gyro→get_zValue()

Returns the angular velocity around the Z axis of the device, as a floating point number.

gyro→isOnline()

Checks if the gyroscope is currently reachable, without raising any error.

gyro→isOnline_async(callback, context)

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

gyro→load(msValidity)

Preloads the gyroscope cache with a specified validity duration.

gyro→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

gyro→load_async(msValidity, callback, context)

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

gyro→nextGyro()

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

gyro→registerAnglesCallback(callback)

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerQuaternionCallback(callback)

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

gyro→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

gyro→set_highestValue(newval)

Changes the recorded maximal value observed.

gyro→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

gyro→set_logicalName(newval)

Changes the logical name of the gyroscope.

gyro→set_lowestValue(newval)

Changes the recorded minimal value observed.

gyro→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

gyro→set_resolution(newval)

Changes the resolution of the measured physical values.

gyro→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

gyro→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGyro.FindGyro() yFindGyro()yFindGyro()

YGyro

Retrieves a gyroscope for a given identifier.

```
function yFindGyro( func: string): TYGyro
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the gyroscope is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGyro.isOnline()` to test if the gyroscope is indeed online at a given time. In case of ambiguity when looking for a gyroscope by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the gyroscope

Returns :

a `YGyro` object allowing you to drive the gyroscope.

YGyro.FirstGyro() yFirstGyro()yFirstGyro()

YGyro

Starts the enumeration of gyroscopes currently accessible.

```
function yFirstGyro( ): TYGyro
```

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

Returns :

a pointer to a `YGyro` object, corresponding to the first gyro currently online, or a `null` pointer if there are none.

gyro→calibrateFromPoints()
gyro.calibrateFromPoints()**YGyro**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                               refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→describe()**gyro.describe()****YGyro**

Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the gyroscope (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

gyro→**get_advertisedValue()**

YGyro

gyro→**advertisedValue()****gyro.get_advertisedValue()**

Returns the current value of the gyroscope (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the gyroscope (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

gyro→get_currentRawValue()**YGyro****gyro→currentRawValue()gyro.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

function **get_currentRawValue()**: double

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

gyro→**get_currentValue()**

YGyro

gyro→**currentValue()****gyro.get_currentValue()**

Returns the current value of the angular velocity.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the angular velocity

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

gyro→**get_errorMessage()****YGyro****gyro**→**errorMessage()****gyro.get_errorMessage()**

Returns the error message of the latest error with the gyroscope.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the gyroscope object

gyro→get_errorType()

YGyro

gyro→errorType()gyro.get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the gyroscope object

gyro→get_functionDescriptor()
gyro→functionDescriptor()
gyro.get_functionDescriptor()

YGyro

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

gyro→**get_heading()**

YGyro

gyro→**heading()****gyro.get_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_heading( ): double
```

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to heading in degrees, between 0 and 360.

gyro→**get_highestValue()****YGyro****gyro**→**highestValue()****gyro.get_highestValue()**

Returns the maximal value observed for the angular velocity since the device was started.

function **get_highestValue()**: double

Returns :

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

gyro→**get_logFrequency()**

YGyro

gyro→**logFrequency()****gyro.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

gyro→**get_logicalName()****YGyro****gyro**→**logicalName()****gyro.get_logicalName()**

Returns the logical name of the gyroscope.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the gyroscope. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

gyro→**get_lowestValue()**

YGyro

gyro→**lowestValue()****gyro.get_lowestValue()**

Returns the minimal value observed for the angular velocity since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

gyro→get_module()**YGyro****gyro→module()gyro.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

gyro→**get_pitch()**

YGyro

gyro→**pitch()****gyro.get_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_pitch( ): double
```

The axis corresponding to the pitch angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

gyro→**get_quaternionW()****YGyro****gyro**→**quaternionW()****gyro.get_quaternionW()**

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_quaternionW( ): double
```

Returns :

a floating-point number corresponding to the w component of the quaternion.

gyro→**get_quaternionX()**

YGyro

gyro→**quaternionX()****gyro.get_quaternionX()**

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_quaternionX( ): double
```

The x component is mostly correlated with rotations on the roll axis.

Returns :

a floating-point number corresponding to the x component of the quaternion.

gyro→**get_quaternionY()****YGyro****gyro**→**quaternionY()****gyro.get_quaternionY()**

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_quaternionY( ): double
```

The y component is mostly correlated with rotations on the pitch axis.

Returns :

a floating-point number corresponding to the y component of the quaternion.

gyro→**get_quaternionZ()**

YGyro

gyro→**quaternionZ()****gyro.get_quaternionZ()**

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_quaternionZ( ): double
```

The x component is mostly correlated with changes of heading.

Returns :

a floating-point number corresponding to the z component of the quaternion.

gyro→get_recordedData()**YGyro****gyro→recordedData()gyro.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

gyro→get_reportFrequency()

YGyro

gyro→reportFrequency()gyro.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

gyro→**get_resolution()****YGyro****gyro**→**resolution()****gyro.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

gyro→**get_roll()**

YGyro

gyro→**roll()****gyro.get_roll()**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
function get_roll( ): double
```

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to roll angle in degrees, between -180 and +180.

gyro→**get_unit()****YGyro****gyro**→**unit()****gyro.get_unit()**

Returns the measuring unit for the angular velocity.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns Y_UNIT_INVALID.

gyro→**get_userData()**

YGyro

gyro→**userData()****gyro.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

gyro→**get_xValue()****YGyro****gyro**→**xValue()****gyro.get_xValue()**

Returns the angular velocity around the X axis of the device, as a floating point number.

function **get_xValue**(): double

Returns :

a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

gyro→**get_yValue()**

YGyro

gyro→**yValue()****gyro.get_yValue()**

Returns the angular velocity around the Y axis of the device, as a floating point number.

```
function get_yValue( ): double
```

Returns :

a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

gyro→**get_zValue()****YGyro****gyro**→**zValue()****gyro.get_zValue()**

Returns the angular velocity around the Z axis of the device, as a floating point number.

function **get_zValue()**: double

Returns :

a floating point number corresponding to the angular velocity around the Z axis of the device, as a floating point number

On failure, throws an exception or returns Y_ZVALUE_INVALID.

gyro→isOnline()gyro.isOnline()

YGyro

Checks if the gyroscope is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

Returns :

`true` if the gyroscope can be reached, and `false` otherwise

gyro→load()**gyro.load()****YGyro**

Preloads the gyroscope cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

gyro→loadCalibrationPoints()
gyro.loadCalibrationPoints()**YGyro**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→nextGyro()gyro.nextGyro()**YGyro**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

```
function nextGyro( ): TYGyro
```

Returns :

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.

gyro→registerAnglesCallback()
gyro.registerAnglesCallback()**YGyro**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
function registerAnglesCallback( callback: TYAnglesCallback): LongInt
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGyro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

gyro→registerQuaternionCallback()
gyro.registerQuaternionCallback()

YGyro

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
function registerQuaternionCallback( callback: TYQuatCallback): LongInt
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

gyro→registerTimedReportCallback()
gyro.registerTimedReportCallback()**YGyro**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYGyroTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

gyro→registerValueCallback()
gyro.registerValueCallback()

YGyro

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYGyroValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

gyro→**set_highestValue()**

YGyro

gyro→**setHighestValue()****gyro.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logFrequency()**YGyro****gyro→setLogFrequency()gyro.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→**set_logicalName()****YGyro****gyro**→**setLogicalName()****gyro.set_logicalName()**

Changes the logical name of the gyroscope.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the gyroscope.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

gyro→set_lowestValue()**YGyro****gyro→setLowestValue()****gyro.set_lowestValue()**

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_reportFrequency()**YGyro****gyro→setReportFrequency()****gyro.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_resolution()**YGyro****gyro→setResolution()gyro.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_userdata()

YGyro

gyro→setUserData()gyro.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.19. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_hubport.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YHubPort = yoctolib.YHubPort;
php	require_once('yocto_hubport.php');
c++	#include "yocto_hubport.h"
m	#import "yocto_hubport.h"
pas	uses yocto_hubport;
vb	yocto_hubport.vb
cs	yocto_hubport.cs
java	import com.yoctopuce.YoctoAPI.YHubPort;
py	from yocto_hubport import *

Global functions

yFindHubPort(func)

Retrieves a Yocto-hub port for a given identifier.

yFirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

YHubPort methods

hubport→describe()

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

hubport→get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

hubport→get_baudRate()

Returns the current baud rate used by this Yocto-hub port, in kbps.

hubport→get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

hubport→get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

hubport→get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

hubport→get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME . FUNCTION_NAME`.

hubport→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

hubport→get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

hubport→get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL . FUNCTIONID`.

hubport→get_logicalName()

Returns the logical name of the Yocto-hub port.

hubport→get_module()

Gets the `YModule` object for the device on which the function is located.

hubport→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

hubport→get_portState()

Returns the current state of the Yocto-hub port.

hubport→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

hubport→isOnline()

Checks if the Yocto-hub port is currently reachable, without raising any error.

hubport→isOnline_async(callback, context)

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

hubport→load(msValidity)

Preloads the Yocto-hub port cache with a specified validity duration.

hubport→load_async(msValidity, callback, context)

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

hubport→nextHubPort()

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

hubport→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

hubport→set_enabled(newval)

Changes the activation of the Yocto-hub port.

hubport→set_logicalName(newval)

Changes the logical name of the Yocto-hub port.

hubport→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

hubport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHubPort.FindHubPort() yFindHubPort()yFindHubPort()

YHubPort

Retrieves a Yocto-hub port for a given identifier.

```
function yFindHubPort( func: string): TYHubPort
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Yocto-hub port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHubPort.isOnline()` to test if the Yocto-hub port is indeed online at a given time. In case of ambiguity when looking for a Yocto-hub port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the Yocto-hub port

Returns :

a `YHubPort` object allowing you to drive the Yocto-hub port.

YHubPort.FirstHubPort() yFirstHubPort()yFirstHubPort()

YHubPort

Starts the enumeration of Yocto-hub ports currently accessible.

```
function yFirstHubPort( ): TYHubPort
```

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

Returns :

a pointer to a `YHubPort` object, corresponding to the first Yocto-hub port currently online, or a `null` pointer if there are none.

hubport→describe()hubport.describe()**YHubPort**

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the Yocto-hub port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

hubport→get_advertisedValue()

YHubPort

hubport→advertisedValue()

hubport.get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the Yocto-hub port (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

hubport→**get_baudRate()****YHubPort****hubport**→**baudRate()****hubport.get_baudRate()**

Returns the current baud rate used by this Yocto-hub port, in kbps.

```
function get_baudRate( ): LongInt
```

The default value is 1000 kbps, but a slower rate may be used if communication problems are encountered.

Returns :

an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns Y_BAUDRATE_INVALID.

hubport→**get_enabled()**

YHubPort

hubport→**enabled()****hubport.get_enabled()**

Returns true if the Yocto-hub port is powered, false otherwise.

function **get_enabled**(): Integer

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the Yocto-hub port is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

hubport→**get_errorMessage()****YHubPort****hubport**→**errorMessage()****hubport.get_errorMessage()**

Returns the error message of the latest error with the Yocto-hub port.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the Yocto-hub port object

hubport→**get_errorType()**

YHubPort

hubport→**errorType()****hubport.get_errorType()**

Returns the numerical error code of the latest error with the Yocto-hub port.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the Yocto-hub port object

hubport→get_functionDescriptor()**YHubPort****hubport→functionDescriptor()****hubport.get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

hubport→**get_logicalName()**

YHubPort

hubport→**logicalName()****hubport.get_logicalName()**

Returns the logical name of the Yocto-hub port.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the Yocto-hub port. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

hubport→**get_module()****YHubPort****hubport**→**module()****hubport.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

hubport→**get_portState()**

YHubPort

hubport→**portState()****hubport.get_portState()**

Returns the current state of the Yocto-hub port.

function **get_portState**(): Integer

Returns :

a value among Y_PORTSTATE_OFF, Y_PORTSTATE_OVRD, Y_PORTSTATE_ON, Y_PORTSTATE_RUN and Y_PORTSTATE_PROG corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns Y_PORTSTATE_INVALID.

hubport→**get_userData()****YHubPort****hubport**→**userData()****hubport.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

hubport→isOnline()hubport.isOnline()

YHubPort

Checks if the Yocto-hub port is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Yocto-hub port.

Returns :

`true` if the Yocto-hub port can be reached, and `false` otherwise

hubport→load()hubport.load()**YHubPort**

Preloads the Yocto-hub port cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

hubport→**nextHubPort()****hubport.nextHubPort()**

YHubPort

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

```
function nextHubPort( ): TYHubPort
```

Returns :

a pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a `null` pointer if there are no more Yocto-hub ports to enumerate.

hubport→registerValueCallback()
hubport.registerValueCallback()

YHubPort

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYHubPortValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

hubport→**set_enabled()****YHubPort****hubport**→**setEnabled()****hubport.set_enabled()**

Changes the activation of the Yocto-hub port.

```
function set_enabled( newval: Integer): integer
```

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the activation of the Yocto-hub port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→set_logicalName()**YHubPort****hubport→setLogicalName()****hubport.set_logicalName()**

Changes the logical name of the Yocto-hub port.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Yocto-hub port.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

hubport→**set_userData()**

YHubPort

hubport→**setUserData()****hubport.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.20. Humidity function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_humidity.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YHumidity = yoctolib.YHumidity;
php	require_once('yocto_humidity.php');
c++	#include "yocto_humidity.h"
m	#import "yocto_humidity.h"
pas	uses yocto_humidity;
vb	yocto_humidity.vb
cs	yocto_humidity.cs
java	import com.yoctopuce.YoctoAPI.YHumidity;
py	from yocto_humidity import *

Global functions

yFindHumidity(func)

Retrieves a humidity sensor for a given identifier.

yFirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

YHumidity methods

humidity→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

humidity→describe()

Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

humidity→get_advertisedValue()

Returns the current value of the humidity sensor (no more than 6 characters).

humidity→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

humidity→get_currentValue()

Returns the current measure for the humidity.

humidity→get_errorMessage()

Returns the error message of the latest error with the humidity sensor.

humidity→get_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

humidity→get_friendlyName()

Returns a global identifier of the humidity sensor in the format `MODULE_NAME . FUNCTION_NAME`.

humidity→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

humidity→get_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

humidity→get_hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form `SERIAL . FUNCTIONID`.

humidity→get_highestValue()

Returns the maximal value observed for the humidity.

humidity→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

Returns the minimal value observed for the humidity.

humidity→get_module()

Gets the YModule object for the device on which the function is located.

humidity→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

humidity→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

humidity→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

humidity→get_resolution()

Returns the resolution of the measured values.

humidity→get_unit()

Returns the measuring unit for the humidity.

humidity→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

humidity→isOnline()

Checks if the humidity sensor is currently reachable, without raising any error.

humidity→isOnline_async(callback, context)

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

humidity→load(msValidity)

Preloads the humidity sensor cache with a specified validity duration.

humidity→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

humidity→load_async(msValidity, callback, context)

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

humidity→nextHumidity()

Continues the enumeration of humidity sensors started using yFirstHumidity().

humidity→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

humidity→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

humidity→set_highestValue(newval)

Changes the recorded maximal value observed for the humidity.

humidity→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

humidity→set_logicalName(newval)

Changes the logical name of the humidity sensor.

humidity→set_lowestValue(newval)

Changes the recorded minimal value observed for the humidity.

humidity→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

humidity→set_resolution(newval)

Changes the resolution of the measured physical values.

humidity→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

humidity→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YHumidity.FindHumidity() yFindHumidity()yFindHumidity()

YHumidity

Retrieves a humidity sensor for a given identifier.

```
function yFindHumidity( func: string): TYHumidity
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the humidity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHumidity.isOnline()` to test if the humidity sensor is indeed online at a given time. In case of ambiguity when looking for a humidity sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the humidity sensor

Returns :

a `YHumidity` object allowing you to drive the humidity sensor.

**YHumidity.FirstHumidity()
yFirstHumidity()yFirstHumidity()**

YHumidity

Starts the enumeration of humidity sensors currently accessible.

```
function yFirstHumidity( ): TYHumidity
```

Use the method `YHumidity.nextHumidity()` to iterate on next humidity sensors.

Returns :

a pointer to a `YHumidity` object, corresponding to the first humidity sensor currently online, or a `null` pointer if there are none.

humidity→calibrateFromPoints() humidity.calibrateFromPoints()

YHumidity

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→describe()humidity.describe()**YHumidity**

Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the humidity sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

humidity→get_advertisedValue()

YHumidity

humidity→advertisedValue()

humidity.get_advertisedValue()

Returns the current value of the humidity sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the humidity sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

humidity→get_currentRawValue()**YHumidity****humidity→currentRawValue()****humidity.get_currentRawValue()**

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

humidity→get_currentValue()

YHumidity

humidity→currentValue()humidity.get_currentValue()

Returns the current measure for the humidity.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current measure for the humidity

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

humidity→get_errorMessage()
humidity→errorMessage()
humidity.get_errorMessage()

YHumidity

Returns the error message of the latest error with the humidity sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the humidity sensor object

humidity→get_errorType()

YHumidity

humidity→errorType()humidity.get_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the humidity sensor object

humidity→get_functionDescriptor()
humidity→functionDescriptor()
humidity.get_functionDescriptor()

YHumidity

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

humidity→get_highestValue()

YHumidity

humidity→highestValue()

humidity.get_highestValue()

Returns the maximal value observed for the humidity.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the humidity

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

humidity→get_logFrequency()**YHumidity****humidity→logFrequency()****humidity.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

humidity→**get_logicalName()**

YHumidity

humidity→**logicalName()****humidity.get_logicalName()**

Returns the logical name of the humidity sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the humidity sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

humidity→get_lowestValue()**YHumidity****humidity→lowestValue()humidity.get_lowestValue()**

Returns the minimal value observed for the humidity.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the humidity

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

humidity→get_module()

YHumidity

humidity→module()humidity.get_module()

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TYModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

humidity→get_recordedData()
humidity→recordedData()
humidity.get_recordedData()

YHumidity

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

humidity→get_reportFrequency()

YHumidity

humidity→reportFrequency()

humidity.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

humidity→get_resolution()**YHumidity****humidity→resolution()humidity.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

humidity→**get_unit()**

YHumidity

humidity→**unit()****humidity.get_unit()**

Returns the measuring unit for the humidity.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

humidity→get_userdata()**YHumidity****humidity→userData()humidity.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

function **get_userdata()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

humidity→isOnline()humidity.isOnline()

YHumidity

Checks if the humidity sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the humidity sensor.

Returns :

true if the humidity sensor can be reached, and false otherwise

humidity→load()humidity.load()**YHumidity**

Preloads the humidity sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

humidity→loadCalibrationPoints()**YHumidity****humidity.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→**nextHumidity()****humidity.nextHumidity()****YHumidity**

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

```
function nextHumidity( ): TYHumidity
```

Returns :

a pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a `null` pointer if there are no more humidity sensors to enumerate.

humidity→registerTimedReportCallback()
humidity.registerTimedReportCallback()**YHumidity**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYHumidityTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

humidity→registerValueCallback()
humidity.registerValueCallback()

YHumidity

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYHumidityValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

humidity→set_highestValue()

YHumidity

humidity→setHighestValue()

humidity.set_highestValue()

Changes the recorded maximal value observed for the humidity.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the humidity

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logFrequency()**YHumidity****humidity→setLogFrequency()****humidity.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logicalName()

YHumidity

humidity→setLogicalName()

humidity.set_logicalName()

Changes the logical name of the humidity sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the humidity sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

humidity→set_lowestValue()**YHumidity****humidity→setLowestValue()****humidity.set_lowestValue()**

Changes the recorded minimal value observed for the humidity.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the humidity

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_reportFrequency()**YHumidity****humidity→setReportFrequency()****humidity.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_resolution()**YHumidity****humidity→setResolution()humidity.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_userdata()

YHumidity

humidity→setUserData()humidity.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.21. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_led.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YLed = yoctolib.YLed;
php	require_once('yocto_led.php');
c++	#include "yocto_led.h"
m	#import "yocto_led.h"
pas	uses yocto_led;
vb	yocto_led.vb
cs	yocto_led.cs
java	import com.yoctopuce.YoctoAPI.YLed;
py	from yocto_led import *

Global functions

yFindLed(func)

Retrieves a led for a given identifier.

yFirstLed()

Starts the enumeration of leds currently accessible.

YLed methods

led→describe()

Returns a short text that describes unambiguously the instance of the led in the form TYPE (NAME) = SERIAL . FUNCTIONID.

led→get_advertisedValue()

Returns the current value of the led (no more than 6 characters).

led→get_blinking()

Returns the current led signaling mode.

led→get_errorMessage()

Returns the error message of the latest error with the led.

led→get_errorType()

Returns the numerical error code of the latest error with the led.

led→get_friendlyName()

Returns a global identifier of the led in the format MODULE_NAME . FUNCTION_NAME.

led→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

led→get_functionId()

Returns the hardware identifier of the led, without reference to the module.

led→get_hardwareId()

Returns the unique hardware identifier of the led in the form SERIAL . FUNCTIONID.

led→get_logicalName()

Returns the logical name of the led.

led→get_luminosity()

Returns the current led intensity (in per cent).

led→get_module()

Gets the `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.

YLed.FindLed()**YLed****yFindLed()**

Retrieves a led for a given identifier.

```
function yFindLed( func: string): TYLed
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the led is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLed.isOnline()` to test if the led is indeed online at a given time. In case of ambiguity when looking for a led by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the led

Returns :

a `YLed` object allowing you to drive the led.

YLed.FirstLed() yFirstLed()yFirstLed()

YLed

Starts the enumeration of leds currently accessible.

```
function yFirstLed( ): TYLed
```

Use the method `YLed.nextLed()` to iterate on next leds.

Returns :

a pointer to a `YLed` object, corresponding to the first led currently online, or a `null` pointer if there are none.

led→describe()led.describe()**YLed**

Returns a short text that describes unambiguously the instance of the led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe(): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

led→**get_advertisedValue()**

YLed

led→**advertisedValue()****led.get_advertisedValue()**

Returns the current value of the led (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the led (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

led→**get_blinking()****YLed****led**→**blinking()****led.get_blinking()**

Returns the current led signaling mode.

function **get_blinking**(): Integer

Returns :

a value among Y_BLINKING_STILL, Y_BLINKING_RELAX, Y_BLINKING_AWARE, Y_BLINKING_RUN, Y_BLINKING_CALL and Y_BLINKING_PANIC corresponding to the current led signaling mode

On failure, throws an exception or returns Y_BLINKING_INVALID.

led→**get_errorMessage()**

YLed

led→**errorMessage()****led.get_errorMessage()**

Returns the error message of the latest error with the led.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the led object

led→**get_errorType()****YLed****led**→**errorType()****led.get_errorType()**

Returns the numerical error code of the latest error with the led.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the led object

led→get_functionDescriptor()

YLed

led→functionDescriptor()

led.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

led→**get_logicalName()****YLed****led**→**logicalName()****led.get_logicalName()**

Returns the logical name of the led.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the led. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

led→**get_luminosity()**

YLed

led→**luminosity()****led.get_luminosity()**

Returns the current led intensity (in per cent).

```
function get_luminosity( ): LongInt
```

Returns :

an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

led→**get_module()****YLed****led**→**module()****led.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): YModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

led→**get_power()**

YLed

led→**power()****led.get_power()**

Returns the current led state.

```
function get_power( ): Integer
```

Returns :

either Y_POWER_OFF or Y_POWER_ON, according to the current led state

On failure, throws an exception or returns Y_POWER_INVALID.

led→**get_userData()****YLed****led**→**userData()****led.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

led→isOnline()led.isOnline()

YLed

Checks if the led is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the led.

Returns :

true if the led can be reached, and false otherwise

led→load()led.load()**YLed**

Preloads the led cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

led→**nextLed()****led.nextLed()**

YLed

Continues the enumeration of leds started using `yFirstLed()`.

```
function nextLed(): TYLed
```

Returns :

a pointer to a `YLed` object, corresponding to a led currently online, or a `null` pointer if there are no more leds to enumerate.

led→registerValueCallback()
led.registerValueCallback()

YLed

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYLedValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

led→**set_blinking()****YLed****led**→**setBlinking()****led.set_blinking()**

Changes the current led signaling mode.

```
function set_blinking( newval: Integer): integer
```

Parameters :

newval a value among Y_BLINKING_STILL, Y_BLINKING_RELAX, Y_BLINKING_AWARE, Y_BLINKING_RUN, Y_BLINKING_CALL and Y_BLINKING_PANIC corresponding to the current led signaling mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→**set_logicalName()****YLed****led**→**setLogicalName()****led.set_logicalName()**

Changes the logical name of the led.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the led.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

led→**set_luminosity()**

YLed

led→**setLuminosity()****led.set_luminosity()**

Changes the current led intensity (in per cent).

```
function set_luminosity( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the current led intensity (in per cent)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→**set_power()****YLed****led**→**setPower()****led.set_power()**

Changes the state of the led.

```
function set_power( newval: Integer): integer
```

Parameters :

newval either Y_POWER_OFF or Y_POWER_ON, according to the state of the led

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→**set_userdata()**

YLed

led→**setUserData()****led.set_userdata()**

Stores a user context provided as argument in the `userData` attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.22. LightSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_lightsensor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YLightSensor = yoctolib.YLightSensor;
php	require_once('yocto_lightsensor.php');
c++	#include "yocto_lightsensor.h"
m	#import "yocto_lightsensor.h"
pas	uses yocto_lightsensor;
vb	yocto_lightsensor.vb
cs	yocto_lightsensor.cs
java	import com.yoctopuce.YoctoAPI.YLightSensor;
py	from yocto_lightsensor import *

Global functions

yFindLightSensor(func)

Retrieves a light sensor for a given identifier.

yFirstLightSensor()

Starts the enumeration of light sensors currently accessible.

YLightSensor methods

lightsensor→calibrate(calibratedVal)

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

lightsensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

lightsensor→describe()

Returns a short text that describes unambiguously the instance of the light sensor in the form TYPE (NAME) = SERIAL . FUNCTIONID.

lightsensor→get_advertisedValue()

Returns the current value of the light sensor (no more than 6 characters).

lightsensor→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

lightsensor→get_currentValue()

Returns the current measure for the ambient light.

lightsensor→get_errorMessage()

Returns the error message of the latest error with the light sensor.

lightsensor→get_errorType()

Returns the numerical error code of the latest error with the light sensor.

lightsensor→get_friendlyName()

Returns a global identifier of the light sensor in the format MODULE_NAME . FUNCTION_NAME.

lightsensor→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

lightsensor→get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

lightsensor→**get_hardwareId()**

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

lightsensor→**get_highestValue()**

Returns the maximal value observed for the ambient light.

lightsensor→**get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

lightsensor→**get_logicalName()**

Returns the logical name of the light sensor.

lightsensor→**get_lowestValue()**

Returns the minimal value observed for the ambient light.

lightsensor→**get_module()**

Gets the `YModule` object for the device on which the function is located.

lightsensor→**get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

lightsensor→**get_recordedData(startTime, endTime)**

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

lightsensor→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

lightsensor→**get_resolution()**

Returns the resolution of the measured values.

lightsensor→**get_unit()**

Returns the measuring unit for the ambient light.

lightsensor→**get_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

lightsensor→**isOnline()**

Checks if the light sensor is currently reachable, without raising any error.

lightsensor→**isOnline_async(callback, context)**

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

lightsensor→**load(msValidity)**

Preloads the light sensor cache with a specified validity duration.

lightsensor→**loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

lightsensor→**load_async(msValidity, callback, context)**

Preloads the light sensor cache with a specified validity duration (asynchronous version).

lightsensor→**nextLightSensor()**

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

lightsensor→**registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

lightsensor→**registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

lightsensor→**set_highestValue(newval)**

Changes the recorded maximal value observed for the ambient light.

lightsensor→**set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

lightsensor→**set_logicalName**(newval)

Changes the logical name of the light sensor.

lightsensor→**set_lowestValue**(newval)

Changes the recorded minimal value observed for the ambient light.

lightsensor→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

lightsensor→**set_resolution**(newval)

Changes the resolution of the measured physical values.

lightsensor→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

lightsensor→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YLightSensor.FindLightSensor() yFindLightSensor()yFindLightSensor()

YLightSensor

Retrieves a light sensor for a given identifier.

```
function yFindLightSensor( func: string): TYLightSensor
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the light sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLightSensor.isOnline()` to test if the light sensor is indeed online at a given time. In case of ambiguity when looking for a light sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the light sensor

Returns :

a `YLightSensor` object allowing you to drive the light sensor.

YLightSensor.FirstLightSensor() yFirstLightSensor()yFirstLightSensor()

YLightSensor

Starts the enumeration of light sensors currently accessible.

```
function yFirstLightSensor( ): TYLightSensor
```

Use the method `YLightSensor.nextLightSensor()` to iterate on next light sensors.

Returns :

a pointer to a `YLightSensor` object, corresponding to the first light sensor currently online, or a `null` pointer if there are none.

lightsensor→calibrate()lightsensor.calibrate()**YLightSensor**

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```
function calibrate( calibratedVal: double): integer
```

Parameters :

calibratedVal the desired target value.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→calibrateFromPoints()
lightsensor.calibrateFromPoints()**YLightSensor**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→describe()lightsensor.describe()**YLightSensor**

Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the light sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

lightsensor→get_advertisedValue()**YLightSensor****lightsensor→advertisedValue()****lightsensor.get_advertisedValue()**

Returns the current value of the light sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the light sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

lightsensor→get_currentRawValue()

YLightSensor

lightsensor→currentRawValue()

lightsensor.get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

lightsensor→get_currentValue()**YLightSensor****lightsensor→currentValue()****lightsensor.get_currentValue()**

Returns the current measure for the ambient light.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current measure for the ambient light

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

lightsensor→get_errorMessage()

YLightSensor

lightsensor→errorMessage()

lightsensor.get_errorMessage()

Returns the error message of the latest error with the light sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the light sensor object

lightsensor→get_errorType()**YLightSensor****lightsensor→errorType()lightsensor.get_errorType()**

Returns the numerical error code of the latest error with the light sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the light sensor object

lightsensor→get_functionDescriptor()
lightsensor→functionDescriptor()
lightsensor.get_functionDescriptor()

YLightSensor

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

lightsensor→get_highestValue()**YLightSensor****lightsensor→highestValue()****lightsensor.get_highestValue()**

Returns the maximal value observed for the ambient light.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the ambient light

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

lightsensor→get_logFrequency()

YLightSensor

lightsensor→logFrequency()

lightsensor.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

lightsensor→get_logicalName()**YLightSensor****lightsensor→logicalName()****lightsensor.get_logicalName()**

Returns the logical name of the light sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the light sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

lightsensor→get_lowestValue()

YLightSensor

lightsensor→lowestValue()

lightsensor.get_lowestValue()

Returns the minimal value observed for the ambient light.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the ambient light

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

lightsensor→get_module()**YLightSensor****lightsensor→module()lightsensor.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

lightsensor→get_recordedData()**YLightSensor****lightsensor→recordedData()****lightsensor.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

lightsensor→get_reportFrequency()**YLightSensor****lightsensor→reportFrequency()****lightsensor.get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

lightsensor→get_resolution()

YLightSensor

lightsensor→resolution()lightsensor.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

lightsensor→**get_unit()****YLightSensor****lightsensor**→**unit()****lightsensor.get_unit()**

Returns the measuring unit for the ambient light.

function **get_unit**(): string

Returns :

a string corresponding to the measuring unit for the ambient light

On failure, throws an exception or returns Y_UNIT_INVALID.

lightsensor→**get_userData()**

YLightSensor

lightsensor→**userData()****lightsensor.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

lightsensor→**isOnline()****lightsensor.isOnline()****YLightSensor**

Checks if the light sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

Returns :

`true` if the light sensor can be reached, and `false` otherwise

lightsensor→load()lightsensor.load()**YLightSensor**

Preloads the light sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

lightsensor→loadCalibrationPoints()
lightsensor.loadCalibrationPoints()**YLightSensor**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→nextLightSensor()
lightsensor.nextLightSensor()

YLightSensor

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

```
function nextLightSensor( ): TYPLightSensor
```

Returns :

a pointer to a `YLightSensor` object, corresponding to a light sensor currently online, or a `null` pointer if there are no more light sensors to enumerate.

lightsensor→registerTimedReportCallback()
lightsensor.registerTimedReportCallback()

YLightSensor

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYLightSensorTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

lightsensor→registerValueCallback()
lightsensor.registerValueCallback()**YLightSensor**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYLightSensorValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

lightsensor→**set_highestValue()****YLightSensor****lightsensor**→**setHighestValue()****lightsensor.set_highestValue()**

Changes the recorded maximal value observed for the ambient light.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the ambient light

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logFrequency()**YLightSensor****lightsensor→setLogFrequency()****lightsensor.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logicalName()**YLightSensor****lightsensor→setLogicalName()****lightsensor.set_logicalName()**

Changes the logical name of the light sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the light sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

lightsensor→**set_lowestValue()****YLightSensor****lightsensor**→**setLowestValue()****lightsensor.set_lowestValue()**

Changes the recorded minimal value observed for the ambient light.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the ambient light

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_reportFrequency()**YLightSensor****lightsensor→setReportFrequency()****lightsensor.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_resolution()

YLightSensor

lightsensor→setResolution()

lightsensor.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_userdata()**YLightSensor****lightsensor→setUserData()****lightsensor.set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.23. Magnetometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_magnetometer.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YMagnetometer = yoctolib.YMagnetometer;
php	require_once('yocto_magnetometer.php');
c++	#include "yocto_magnetometer.h"
m	#import "yocto_magnetometer.h"
pas	uses yocto_magnetometer;
vb	yocto_magnetometer.vb
cs	yocto_magnetometer.cs
java	import com.yoctopuce.YoctoAPI.YMagnetometer;
py	from yocto_magnetometer import *

Global functions

yFindMagnetometer(func)

Retrieves a magnetometer for a given identifier.

yFirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

YMagnetometer methods

magnetometer→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

magnetometer→describe()

Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

magnetometer→get_advertisedValue()

Returns the current value of the magnetometer (no more than 6 characters).

magnetometer→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

magnetometer→get_currentValue()

Returns the current value of the magnetic field.

magnetometer→get_errorMessage()

Returns the error message of the latest error with the magnetometer.

magnetometer→get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

magnetometer→get_friendlyName()

Returns a global identifier of the magnetometer in the format `MODULE_NAME . FUNCTION_NAME`.

magnetometer→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

magnetometer→get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

magnetometer→get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form `SERIAL . FUNCTIONID`.

magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

magnetometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

magnetometer→get_logicalName()

Returns the logical name of the magnetometer.

magnetometer→get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

magnetometer→get_module()

Gets the YModule object for the device on which the function is located.

magnetometer→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

magnetometer→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

magnetometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

magnetometer→get_resolution()

Returns the resolution of the measured values.

magnetometer→get_unit()

Returns the measuring unit for the magnetic field.

magnetometer→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

magnetometer→get_xValue()

Returns the X component of the magnetic field, as a floating point number.

magnetometer→get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

magnetometer→get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

magnetometer→isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

magnetometer→isOnline_async(callback, context)

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

magnetometer→load(msValidity)

Preloads the magnetometer cache with a specified validity duration.

magnetometer→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

magnetometer→load_async(msValidity, callback, context)

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

magnetometer→nextMagnetometer()

Continues the enumeration of magnetometers started using yFirstMagnetometer().

magnetometer→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

magnetometer→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

3. Reference

magnetometer→set_highestValue(newval)

Changes the recorded maximal value observed.

magnetometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

magnetometer→set_logicalName(newval)

Changes the logical name of the magnetometer.

magnetometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

magnetometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

magnetometer→set_resolution(newval)

Changes the resolution of the measured physical values.

magnetometer→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

magnetometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YMagnetometer.FindMagnetometer() yFindMagnetometer()yFindMagnetometer()

YMagnetometer

Retrieves a magnetometer for a given identifier.

```
function yFindMagnetometer( func: string): TYMagnetometer
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMagnetometer.isOnline()` to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the magnetometer

Returns :

a `YMagnetometer` object allowing you to drive the magnetometer.

YMagnetometer.FirstMagnetometer() yFirstMagnetometer()yFirstMagnetometer()

YMagnetometer

Starts the enumeration of magnetometers currently accessible.

```
function yFirstMagnetometer( ): TYMagnetometer
```

Use the method `YMagnetometer.nextMagnetometer()` to iterate on next magnetometers.

Returns :

a pointer to a `YMagnetometer` object, corresponding to the first magnetometer currently online, or a `null` pointer if there are none.

magnetometer→calibrateFromPoints()
magnetometer.calibrateFromPoints()

YMagnetometer

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→describe()**magnetometer.describe()****YMagnetometer**

Returns a short text that describes unambiguously the instance of the magnetometer in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the magnetometer (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

magnetometer→**get_advertisedValue()****YMagnetometer****magnetometer**→**advertisedValue()****magnetometer.get_advertisedValue()**

Returns the current value of the magnetometer (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the magnetometer (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

magnetometer→get_currentRawValue()

YMagnetometer

magnetometer→currentRawValue()

magnetometer.get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

magnetometer→get_currentValue()**YMagnetometer****magnetometer→currentValue()****magnetometer.get_currentValue()**

Returns the current value of the magnetic field.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the magnetic field

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

magnetometer→get_errorMessage()

YMagnetometer

magnetometer→errorMessage()

magnetometer.get_errorMessage()

Returns the error message of the latest error with the magnetometer.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the magnetometer object

magnetometer→get_errorType()
magnetometer→errorType()
magnetometer.get_errorType()

YMagnetometer

Returns the numerical error code of the latest error with the magnetometer.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the magnetometer object

magnetometer→get_functionDescriptor()

YMagnetometer

magnetometer→functionDescriptor()

magnetometer.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

magnetometer→**get_highestValue()****YMagnetometer****magnetometer**→**highestValue()****magnetometer.get_highestValue()**

Returns the maximal value observed for the magnetic field since the device was started.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

magnetometer→get_logFrequency()

YMagnetometer

magnetometer→logFrequency()

magnetometer.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

magnetometer→get_logicalName()**YMagnetometer****magnetometer→logicalName()****magnetometer.get_logicalName()**

Returns the logical name of the magnetometer.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the magnetometer. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

magnetometer→get_lowestValue()

YMagnetometer

magnetometer→lowestValue()

magnetometer.get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

magnetometer→get_module()
magnetometer→module()
magnetometer.get_module()

YMagnetometer

Gets the YModule object for the device on which the function is located.

```
function get_module( ): YModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

magnetometer→**get_recordedData()****YMagnetometer****magnetometer**→**recordedData()****magnetometer.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

magnetometer→get_reportFrequency()**YMagnetometer****magnetometer→reportFrequency()****magnetometer.get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

magnetometer→get_resolution()

YMagnetometer

magnetometer→resolution()

magnetometer.get_resolution()

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

magnetometer→**get_unit()****YMagnetometer****magnetometer**→**unit()****magnetometer.get_unit()**

Returns the measuring unit for the magnetic field.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns Y_UNIT_INVALID.

magnetometer→get_userdata()

YMagnetometer

magnetometer→userData()

magnetometer.get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
function get_userdata( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

magnetometer→**get_xValue()****YMagnetometer****magnetometer**→**xValue()****magnetometer.get_xValue()**

Returns the X component of the magnetic field, as a floating point number.

function **get_xValue()**: double

Returns :

a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

magnetometer→get_yValue()

YMagnetometer

magnetometer→yValue()magnetometer.get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

```
function get_yValue( ): double
```

Returns :

a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_YVALUE_INVALID.

magnetometer→**get_zValue()****YMagnetometer****magnetometer**→**zValue()****magnetometer.get_zValue()**

Returns the Z component of the magnetic field, as a floating point number.

function **get_zValue()**: double

Returns :

a floating point number corresponding to the Z component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_ZVALUE_INVALID.

magnetometer→isOnline()magnetometer.isOnline()

YMagnetometer

Checks if the magnetometer is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the magnetometer.

Returns :

`true` if the magnetometer can be reached, and `false` otherwise

magnetometer→load()magnetometer.load()**YMagnetometer**

Preloads the magnetometer cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

magnetometer→loadCalibrationPoints()**YMagnetometer****magnetometer.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→**nextMagnetometer()**
magnetometer.nextMagnetometer()

YMagnetometer

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

```
function nextMagnetometer( ): TYMagnetometer
```

Returns :

a pointer to a `YMagnetometer` object, corresponding to a magnetometer currently online, or a `null` pointer if there are no more magnetometers to enumerate.

**magnetometer→registerTimedReportCallback()
magnetometer.registerTimedReportCallback()****YMagnetometer**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYMagnetometerTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

magnetometer→registerValueCallback()
magnetometer.registerValueCallback()

YMagnetometer

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYMagnetometerValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

magnetometer→set_highestValue()

YMagnetometer

magnetometer→setHighestValue()

magnetometer.set_highestValue()

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logFrequency()
magnetometer→setLogFrequency()
magnetometer.set_logFrequency()

YMagnetometer

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logicalName()

YMagnetometer

magnetometer→setLogicalName()

magnetometer.set_logicalName()

Changes the logical name of the magnetometer.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the magnetometer.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

magnetometer→set_lowestValue()**YMagnetometer****magnetometer→setLowestValue()****magnetometer.set_lowestValue()**

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_reportFrequency()

YMagnetometer

magnetometer→setReportFrequency()

magnetometer.set_reportFrequency()

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_resolution()**YMagnetometer****magnetometer→setResolution()****magnetometer.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_userdata()

YMagnetometer

magnetometer→setUserData()

magnetometer.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.24. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

YMeasure methods

measure→get_averageValue()

Returns the average value observed during the time interval covered by this measure.

measure→get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_maxValue()

Returns the largest value observed during the time interval covered by this measure.

measure→get_minValue()

Returns the smallest value observed during the time interval covered by this measure.

measure→get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→**get_averageValue()**

YMeasure

measure→**averageValue()**

measure.get_averageValue()

Returns the average value observed during the time interval covered by this measure.

```
function get_averageValue( ): double
```

Returns :

a floating-point number corresponding to the average value observed.

measure→**get_endTimeUTC()****YMeasure****measure**→**endTimeUTC()****measure.get_endTimeUTC()**

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

function **get_endTimeUTC**(): double

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→**get_maxValue()**

YMeasure

measure→**maxValue()****measure.get_maxValue()**

Returns the largest value observed during the time interval covered by this measure.

function **get_maxValue**(): double

Returns :

a floating-point number corresponding to the largest value observed.

measure→**get_minValue()****YMeasure****measure**→**minValue()****measure.get_minValue()**

Returns the smallest value observed during the time interval covered by this measure.

function **get_minValue**(): double

Returns :

a floating-point number corresponding to the smallest value observed.

measure→**get_startTimeUTC()**

YMeasure

measure→**startTimeUTC()**

measure.get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
function get_startTimeUTC( ): double
```

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

3.25. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
c++	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

Global functions

yFindModule(func)

Allows you to find a module from its serial number or from its logical name.

yFirstModule()

Starts the enumeration of modules currently accessible.

YModule methods

module→describe()

Returns a descriptive text that identifies the module.

module→download(pathname)

Downloads the specified built-in file and returns a binary buffer with its content.

module→functionCount()

Returns the number of functions (beside the "module" interface) available on the module.

module→functionId(functionIndex)

Retrieves the hardware identifier of the *n*th function on the module.

module→functionName(functionIndex)

Retrieves the logical name of the *n*th function on the module.

module→functionValue(functionIndex)

Retrieves the advertised value of the *n*th function on the module.

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

Returns the error message of the 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()

3. Reference

	Returns the icon of the module.
module → get_lastLogs()	Returns a string with last logs of the module.
module → get_logicalName()	Returns the logical name of the module.
module → get_luminosity()	Returns the luminosity of the module informative leds (from 0 to 100).
module → get_persistentSettings()	Returns the current state of persistent module settings.
module → get_productId()	Returns the USB device identifier of the module.
module → get_productName()	Returns the commercial name of the module, as set by the factory.
module → get_productRelease()	Returns the hardware release version of the module.
module → get_rebootCountdown()	Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
module → get_serialNumber()	Returns the serial number of the module, as set by the factory.
module → get_upTime()	Returns the number of milliseconds spent since the module was powered on.
module → get_usbBandwidth()	Returns the number of USB interfaces used by the module.
module → get_usbCurrent()	Returns the current consumed by the module on the USB bus, in milli-amps.
module → get_userData()	Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
module → isOnline()	Checks if the module is currently reachable, without raising any error.
module → isOnline_async(callback, context)	Checks if the module is currently reachable, without raising any error.
module → load(msValidity)	Preloads the module cache with a specified validity duration.
module → load_async(msValidity, callback, context)	Preloads the module cache with a specified validity duration (asynchronous version).
module → nextModule()	Continues the module enumeration started using <code>yFirstModule()</code> .
module → reboot(secBeforeReboot)	Schedules a simple module reboot after the given number of seconds.
module → registerLogCallback(callback)	todo
module → revertFromFlash()	Reloads the settings stored in the nonvolatile memory, as when the module is powered on.
module → saveToFlash()	Saves current settings in the nonvolatile memory of the module.

module→set_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_usbBandwidth(newval)

Changes the number of USB interfaces used by the module.

module→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YModule.FindModule() yFindModule()yFindModule()

YModule

Allows you to find a module from its serial number or from its logical name.

```
function yFindModule( func: string): TYModule
```

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string containing either the serial number or the logical name of the desired module

Returns :

a `YModule` object allowing you to drive the module or get additional information on the module.

**YModule.FirstModule()
yFirstModule()yFirstModule()**

YModule

Starts the enumeration of modules currently accessible.

```
function yFirstModule( ): TYModule
```

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

module→**describe()****module.describe()**

YModule

Returns a descriptive text that identifies the module.

function **describe**(): string

The text may include either the logical name or the serial number of the module.

Returns :

a string that describes the module

module→**download()****module.download()****YModule**

Downloads the specified built-in file and returns a binary buffer with its content.

```
function download( pathname: string): TByteArray
```

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

module→**functionCount()****module.functionCount()**

YModule

Returns the number of functions (beside the "module" interface) available on the module.

function **functionCount**(): integer

Returns :

the number of functions on the module

On failure, throws an exception or returns a negative error code.

module→**functionId()****module.functionId()****YModule**

Retrieves the hardware identifier of the n th function on the module.

```
function functionId( functionIndex: integer): string
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

module→**functionName()****module.functionName()****YModule**

Retrieves the logical name of the n th function on the module.

```
function functionName( functionIndex: integer): string
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

module→**functionValue()****module.functionValue()****YModule**

Retrieves the advertised value of the *n*th function on the module.

```
function functionValue( functionIndex: integer): string
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

module→**get_beacon()**

YModule

module→**beacon()****module.get_beacon()**

Returns the state of the localization beacon.

function **get_beacon**(): Integer

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module→**get_errorMessage()****YModule****module**→**errorMessage()****module.errorMessage()**

Returns the error message of the latest error with this module object.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this module object

module→**get_errorType()**

YModule

module→**errorType()****module.get_errorType()**

Returns the numerical error code of the latest error with this module object.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using this module object

module→**get_firmwareRelease()**
module→**firmwareRelease()**
module.get_firmwareRelease()

YModule

Returns the version of the firmware embedded in the module.

```
function get_firmwareRelease( ): string
```

Returns :

a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

module→**get_icon2d()**

YModule

module→**icon2d()****module.get_icon2d()**

Returns the icon of the module.

```
function get_icon2d( ): TByteArray
```

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format.

module→**get_lastLogs()****YModule****module**→**lastLogs()****module.get_lastLogs()**

Returns a string with last logs of the module.

```
function get_lastLogs( ): string
```

This method return only logs that are still in the module.

Returns :

a string with last logs of the module.

module→**get_logicalName()**

YModule

module→**logicalName()****module.get_logicalName()**

Returns the logical name of the module.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the module

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

module→**get_luminosity()****YModule****module**→**luminosity()****module.get_luminosity()**

Returns the luminosity of the module informative leds (from 0 to 100).

function **get_luminosity()**: LongInt

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

module→**get_persistentSettings()**
module→**persistentSettings()**
module.get_persistentSettings()

YModule

Returns the current state of persistent module settings.

function **get_persistentSettings**(): Integer

Returns :

a value among Y_PERSISTENTSETTINGS_LOADED, Y_PERSISTENTSETTINGS_SAVED and Y_PERSISTENTSETTINGS_MODIFIED corresponding to the current state of persistent module settings

On failure, throws an exception or returns Y_PERSISTENTSETTINGS_INVALID.

module→**get_productId()****YModule****module**→**productId()****module.get_productId()**

Returns the USB device identifier of the module.

function **get_productId**(): LongInt

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y_PRODUCTID_INVALID.

module→**get_productName()**

YModule

module→**productName()****module.get_productName()**

Returns the commercial name of the module, as set by the factory.

```
function get_productName( ): string
```

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y_PRODUCTNAME_INVALID.

module→get_productRelease()
module→productRelease()
module.get_productRelease()

YModule

Returns the hardware release version of the module.

```
function get_productRelease( ): LongInt
```

Returns :

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.

module→get_rebootCountdown()

YModule

module→rebootCountdown()

module.get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

function **get_rebootCountdown()** : LongInt

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns Y_REBOOTCOUNTDOWN_INVALID.

module→**get_serialNumber()****YModule****module**→**serialNumber()****module.get_serialNumber()**

Returns the serial number of the module, as set by the factory.

```
function get_serialNumber( ): string
```

Returns :

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns `Y_SERIALNUMBER_INVALID`.

module→**get_upTime()**

YModule

module→**upTime()****module.get_upTime()**

Returns the number of milliseconds spent since the module was powered on.

```
function get_upTime( ): int64
```

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `Y_UPTIME_INVALID`.

module→**get_usbBandwidth()**
module→**usbBandwidth()**
module.get_usbBandwidth()

YModule

Returns the number of USB interfaces used by the module.

function **get_usbBandwidth**(): Integer

Returns :

either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

On failure, throws an exception or returns `Y_USBBANDWIDTH_INVALID`.

module→**get_usbCurrent()**

YModule

module→**usbCurrent()****module.get_usbCurrent()**

Returns the current consumed by the module on the USB bus, in milli-amps.

function **get_usbCurrent**(): LongInt

Returns :

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

module→**get_userdata()****YModule****module**→**userData()****module.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

function **get_userdata()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

module→isOnline()module.isOnline()

YModule

Checks if the module is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

`true` if the module can be reached, and `false` otherwise

module→load()module.load()

YModule

Preloads the module cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→**nextModule()****module.nextModule()**

YModule

Continues the module enumeration started using `yFirstModule()`.

function **nextModule**(): TYModule

Returns :

a pointer to a YModule object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot()module.reboot()

YModule

Schedules a simple module reboot after the given number of seconds.

```
function reboot( secBeforeReboot: LongInt): LongInt
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→revertFromFlash()
module.revertFromFlash()

YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

function **revertFromFlash**(): LongInt

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→saveToFlash()module.saveToFlash()

YModule

Saves current settings in the nonvolatile memory of the module.

```
function saveToFlash( ): LongInt
```

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

module→**set_beacon()****YModule****module**→**setBeacon()****module.set_beacon()**

Turns on or off the module localization beacon.

```
function set_beacon( newval: Integer): integer
```

Parameters :

newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_logicalName()****YModule****module**→**setLogicalName()****module.set_logicalName()**

Changes the logical name of the module.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_luminosity()****YModule****module**→**setLuminosity()****module.set_luminosity()**

Changes the luminosity of the module informative leds.

```
function set_luminosity( newval: LongInt): integer
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_usbBandwidth()**
module→**setUsbBandwidth()**
module.set_usbBandwidth()

YModule

Changes the number of USB interfaces used by the module.

```
function set_usbBandwidth( newval: Integer): integer
```

You must reboot the module after changing this setting.

Parameters :

newval either Y_USBBANDWIDTH_SIMPLE or Y_USBBANDWIDTH_DOUBLE, according to the number of USB interfaces used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_userData()**

YModule

module→**setUserData()****module.set_userData()**

Stores a user context provided as argument in the `userData` attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

module→triggerFirmwareUpdate()
module.triggerFirmwareUpdate()

YModule

Schedules a module reboot into special firmware update mode.

```
function triggerFirmwareUpdate( secBeforeReboot: LongInt): LongInt
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

3.26. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_network.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YNetwork = yoctolib.YNetwork;
php	require_once('yocto_network.php');
c++	#include "yocto_network.h"
m	#import "yocto_network.h"
pas	uses yocto_network;
vb	yocto_network.vb
cs	yocto_network.cs
java	import com.yoctopuce.YoctoAPI.YNetwork;
py	from yocto_network import *

Global functions

yFindNetwork(func)

Retrieves a network interface for a given identifier.

yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

YNetwork methods

network→callbackLogin(username, password)

Connects to the notification callback and saves the credentials required to log into it.

network→describe()

Returns a short text that describes unambiguously the instance of the network interface in the form TYPE (NAME) = SERIAL . FUNCTIONID.

network→get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

network→get_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

network→get_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

network→get_callbackEncoding()

Returns the encoding standard to use for representing notification values.

network→get_callbackMaxDelay()

Returns the maximum waiting time between two callback notifications, in seconds.

network→get_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

network→get_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

network→get_callbackUrl()

Returns the callback URL to notify of significant state changes.

network→get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→get_errorMessage()

Returns the error message of the latest error with the network interface.

network→get_errorType()

Returns the numerical error code of the latest error with the network interface.

network→get_friendlyName()

Returns a global identifier of the network interface in the format `MODULE_NAME . FUNCTION_NAME`.

network→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

network→get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

network→get_hardwareId()

Returns the unique hardware identifier of the network interface in the form `SERIAL . FUNCTIONID`.

network→get_ipAddress()

Returns the IP address currently in use by the device.

network→get_logicalName()

Returns the logical name of the network interface.

network→get_macAddress()

Returns the MAC address of the network interface.

network→get_module()

Gets the `YModule` object for the device on which the function is located.

network→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

network→get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

network→get_primaryDNS()

Returns the IP address of the primary name server to be used by the module.

network→get_readiness()

Returns the current established working mode of the network interface.

network→get_router()

Returns the IP address of the router on the device subnet (default gateway).

network→get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

network→get_subnetMask()

Returns the subnet mask currently used by the device.

network→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

network→get_userPassword()

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

network→get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→isOnline()

Checks if the network interface is currently reachable, without raising any error.

network→isOnline_async(callback, context)

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

network→load(msValidity)

Preloads the network interface cache with a specified validity duration.

network→load_async(msValidity, callback, context)

Preloads the network interface cache with a specified validity duration (asynchronous version).

network→nextNetwork()

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

network→ping(host)

Pings `str_host` to test the network connectivity.

network→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

network→set_adminPassword(newval)

Changes the password for the "admin" user.

network→set_callbackCredentials(newval)

Changes the credentials required to connect to the callback address.

network→set_callbackEncoding(newval)

Changes the encoding standard to use for representing notification values.

network→set_callbackMaxDelay(newval)

Changes the maximum waiting time between two callback notifications, in seconds.

network→set_callbackMethod(newval)

Changes the HTTP method used to notify callbacks for significant state changes.

network→set_callbackMinDelay(newval)

Changes the minimum waiting time between two callback notifications, in seconds.

network→set_callbackUrl(newval)

Changes the callback URL to notify significant state changes.

network→set_discoverable(newval)

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→set_logicalName(newval)

Changes the logical name of the network interface.

network→set_primaryDNS(newval)

Changes the IP address of the primary name server to be used by the module.

network→set_secondaryDNS(newval)

Changes the IP address of the secondary name server to be used by the module.

network→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

network→set_userPassword(newval)

Changes the password for the "user" user.

network→set_wwwWatchdogDelay(newval)

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

network→useStaticIP(ipAddress, subnetMaskLen, router)

Changes the configuration of the network interface to use a static IP address.

network→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YNetwork.FindNetwork() yFindNetwork()yFindNetwork()

YNetwork

Retrieves a network interface for a given identifier.

```
function yFindNetwork( func: string): TYNetwork
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the network interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YNetwork.IsOnline()` to test if the network interface is indeed online at a given time. In case of ambiguity when looking for a network interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the network interface

Returns :

a `YNetwork` object allowing you to drive the network interface.

**YNetwork.FirstNetwork()
yFirstNetwork()yFirstNetwork()**

YNetwork

Starts the enumeration of network interfaces currently accessible.

```
function yFirstNetwork( ): TYNetwork
```

Use the method `YNetwork.nextNetwork()` to iterate on next network interfaces.

Returns :

a pointer to a `YNetwork` object, corresponding to the first network interface currently online, or a `null` pointer if there are none.

network→callbackLogin()network.callbackLogin()**YNetwork**

Connects to the notification callback and saves the credentials required to log into it.

```
function callbackLogin( username: string, password: string): integer
```

The password is not stored into the module, only a hashed copy of the credentials are saved. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

username username required to log to the callback

password password required to log to the callback

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→describe()network.describe()**YNetwork**

Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the network interface (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

network→get_adminPassword()

YNetwork

network→adminPassword()

network.get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

```
function get_adminPassword( ): string
```

Returns :

a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns Y_ADMINPASSWORD_INVALID.

network→get_advertisedValue()**YNetwork****network→advertisedValue()****network.get_advertisedValue()**

Returns the current value of the network interface (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the network interface (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

network→get_callbackCredentials()

YNetwork

network→callbackCredentials()

network.get_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

```
function get_callbackCredentials( ): string
```

Returns :

a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns Y_CALLBACKCREDENTIALS_INVALID.

network→get_callbackEncoding()**YNetwork****network→callbackEncoding()****network.get_callbackEncoding()**

Returns the encoding standard to use for representing notification values.

```
function get_callbackEncoding( ): Integer
```

Returns :

a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV and Y_CALLBACKENCODING_YOCTO_API corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns Y_CALLBACKENCODING_INVALID.

network→get_callbackMaxDelay()

YNetwork

network→callbackMaxDelay()

network.get_callbackMaxDelay()

Returns the maximum waiting time between two callback notifications, in seconds.

```
function get_callbackMaxDelay( ): LongInt
```

Returns :

an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMAXDELAY_INVALID.

network→get_callbackMethod()**YNetwork****network→callbackMethod()****network.get_callbackMethod()**

Returns the HTTP method used to notify callbacks for significant state changes.

```
function get_callbackMethod( ): Integer
```

Returns :

a value among Y_CALLBACKMETHOD_POST, Y_CALLBACKMETHOD_GET and Y_CALLBACKMETHOD_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns Y_CALLBACKMETHOD_INVALID.

network→get_callbackMinDelay()

YNetwork

network→callbackMinDelay()

network.get_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

function **get_callbackMinDelay**(): LongInt

Returns :

an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y_CALLBACKMINDELAY_INVALID.

network→get_callbackUrl()**YNetwork****network→callbackUrl()network.get_callbackUrl()**

Returns the callback URL to notify of significant state changes.

```
function get_callbackUrl( ): string
```

Returns :

a string corresponding to the callback URL to notify of significant state changes

On failure, throws an exception or returns Y_CALLBACKURL_INVALID.

network→**get_discoverable()****YNetwork****network**→**discoverable()****network.get_discoverable()**

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
function get_discoverable( ): Integer
```

Returns :

either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns `Y_DISCOVERABLE_INVALID`.

network→**get_errorMessage()****YNetwork****network**→**errorMessage()****network.get_errorMessage()**

Returns the error message of the latest error with the network interface.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the network interface object

network→**get_errorType()**

YNetwork

network→**errorType()****network.get_errorType()**

Returns the numerical error code of the latest error with the network interface.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the network interface object

network→get_functionDescriptor()
network→functionDescriptor()
network.get_functionDescriptor()

YNetwork

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

network→**get_ipAddress()**

YNetwork

network→**ipAddress()****network.get_ipAddress()**

Returns the IP address currently in use by the device.

```
function get_ipAddress( ): string
```

The address may have been configured statically, or provided by a DHCP server.

Returns :

a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns `Y_IPADDRESS_INVALID`.

network→**get_logicalName()****YNetwork****network**→**logicalName()****network.get_logicalName()**

Returns the logical name of the network interface.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the network interface. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

network→**get_macAddress()**

YNetwork

network→**macAddress()****network.get_macAddress()**

Returns the MAC address of the network interface.

```
function get_macAddress( ): string
```

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

Returns :

a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns Y_MACADDRESS_INVALID.

network→get_module()**YNetwork****network→module()network.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

network→**get_poeCurrent()**

YNetwork

network→**poeCurrent()****network.get_poeCurrent()**

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

```
function get_poeCurrent( ): LongInt
```

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

Returns :

an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns `Y_POECURRENT_INVALID`.

network→**get_primaryDNS()****YNetwork****network**→**primaryDNS()****network.get_primaryDNS()**

Returns the IP address of the primary name server to be used by the module.

```
function get_primaryDNS( ): string
```

Returns :

a string corresponding to the IP address of the primary name server to be used by the module

On failure, throws an exception or returns Y_PRIMARYDNS_INVALID.

network→**get_readiness()****YNetwork****network**→**readiness()****network.get_readiness()**

Returns the current established working mode of the network interface.

function **get_readiness**(): Integer

Level zero (DOWN_0) means that no hardware link has been detected. Either there is no signal on the network cable, or the selected wireless access point cannot be detected. Level 1 (LIVE_1) is reached when the network is detected, but is not yet connected. For a wireless network, this shows that the requested SSID is present. Level 2 (LINK_2) is reached when the hardware connection is established. For a wired network connection, level 2 means that the cable is attached at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS_4) is reached when the DNS server is reachable on the network. Level 5 (WWW_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

Returns :

a value among Y_READINESS_DOWN, Y_READINESS_EXISTS, Y_READINESS_LINKED, Y_READINESS_LAN_OK and Y_READINESS_WWW_OK corresponding to the current established working mode of the network interface

On failure, throws an exception or returns Y_READINESS_INVALID.

network→get_router()**YNetwork****network→router()network.get_router()**

Returns the IP address of the router on the device subnet (default gateway).

```
function get_router( ): string
```

Returns :

a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns Y_ROUTER_INVALID.

network→get_secondaryDNS()

YNetwork

network→secondaryDNS()

network.get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

```
function get_secondaryDNS( ): string
```

Returns :

a string corresponding to the IP address of the secondary name server to be used by the module

On failure, throws an exception or returns Y_SECONDARYDNS_INVALID.

network→**get_subnetMask()****YNetwork****network**→**subnetMask()****network.get_subnetMask()**

Returns the subnet mask currently used by the device.

```
function get_subnetMask( ): string
```

Returns :

a string corresponding to the subnet mask currently used by the device

On failure, throws an exception or returns Y_SUBNETMASK_INVALID.

network→**get_userData()**

YNetwork

network→**userData()****network.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

network→get_userPassword()**YNetwork****network→userPassword()****network.get_userPassword()**

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

```
function get_userPassword( ): string
```

Returns :

a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns Y_USERPASSWORD_INVALID.

network→**get_wwwWatchdogDelay()****YNetwork****network**→**wwwWatchdogDelay()****network.get_wwwWatchdogDelay()**

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
function get_wwwWatchdogDelay( ): LongInt
```

A zero value disables automated reboot in case of Internet connectivity loss.

Returns :

an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

On failure, throws an exception or returns Y_WWWWATCHDOGDELAY_INVALID.

network→isOnline()network.isOnline()**YNetwork**

Checks if the network interface is currently reachable, without raising any error.

```
function isOnline(): boolean
```

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

Returns :

`true` if the network interface can be reached, and `false` otherwise

network→load()network.load()**YNetwork**

Preloads the network interface cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

network→**nextNetwork()****network.nextNetwork()****YNetwork**

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

```
function nextNetwork( ): TNetwork
```

Returns :

a pointer to a `YNetwork` object, corresponding to a network interface currently online, or a `null` pointer if there are no more network interfaces to enumerate.

network→ping()network.ping()**YNetwork**

Pings str_host to test the network connectivity.

```
function ping( host: string): string
```

Sends four ICMP ECHO_REQUEST requests from the module to the target str_host. This method returns a string with the result of the 4 ICMP ECHO_REQUEST requests.

Parameters :

host the hostname or the IP address of the target

Returns :

a string with the result of the ping.

network→registerValueCallback()
network.registerValueCallback()

YNetwork

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYNetworkValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

network→**set_adminPassword()**
network→**setAdminPassword()**
network.set_adminPassword()

YNetwork

Changes the password for the "admin" user.

```
function set_adminPassword( newval: string): integer
```

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "admin" user

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackCredentials()**YNetwork****network→setCallbackCredentials()****network.set_callbackCredentials()**

Changes the credentials required to connect to the callback address.

```
function set_callbackCredentials( newval: string): integer
```

The credentials must be provided as returned by function `get_callbackCredentials`, in the form `username:hash`. The method used to compute the hash varies according to the authentication scheme implemented by the callback, For Basic authentication, the hash is the MD5 of the string `username:password`. For Digest authentication, the hash is the MD5 of the string `username:realm:password`. For a simpler way to configure callback credentials, use function `callbackLogin` instead. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the credentials required to connect to the callback address

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackEncoding()****YNetwork****network**→**setCallbackEncoding()****network.set_callbackEncoding()**

Changes the encoding standard to use for representing notification values.

```
function set_callbackEncoding( newval: Integer): integer
```

Parameters :

newval a value among Y_CALLBACKENCODING_FORM, Y_CALLBACKENCODING_JSON, Y_CALLBACKENCODING_JSON_ARRAY, Y_CALLBACKENCODING_CSV and Y_CALLBACKENCODING_YOCTO_API corresponding to the encoding standard to use for representing notification values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackMaxDelay()****YNetwork****network**→**setCallbackMaxDelay()****network.set_callbackMaxDelay()**

Changes the maximum waiting time between two callback notifications, in seconds.

```
function set_callbackMaxDelay( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the maximum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackMethod()****YNetwork****network**→**setCallbackMethod()****network.set_callbackMethod()**

Changes the HTTP method used to notify callbacks for significant state changes.

```
function set_callbackMethod( newval: Integer): integer
```

Parameters :

newval a value among Y_CALLBACKMETHOD_POST, Y_CALLBACKMETHOD_GET and Y_CALLBACKMETHOD_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMinDelay()**YNetwork****network→setCallbackMinDelay()****network.set_callbackMinDelay()**

Changes the minimum waiting time between two callback notifications, in seconds.

```
function set_callbackMinDelay( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the minimum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_callbackUrl()****YNetwork****network**→**setCallbackUrl()****network.set_callbackUrl()**

Changes the callback URL to notify significant state changes.

```
function set_callbackUrl( newval: string): integer
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the callback URL to notify significant state changes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_discoverable()**YNetwork****network→setDiscoverable()****network.set_discoverable()**

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
function set_discoverable( newval: Integer): integer
```

Parameters :

newval either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_logicalName()****YNetwork****network**→**setLogicalName()****network.set_logicalName()**

Changes the logical name of the network interface.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the network interface.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

network→set_primaryDNS()**YNetwork****network→setPrimaryDNS()network.set_primaryDNS()**

Changes the IP address of the primary name server to be used by the module.

```
function set_primaryDNS( newval: string): integer
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the primary name server to be used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→**set_secondaryDNS()****YNetwork****network**→**setSecondaryDNS()****network.set_secondaryDNS()**

Changes the IP address of the secondary name server to be used by the module.

```
function set_secondaryDNS( newval: string): integer
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the secondary name server to be used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_userdata()**YNetwork****network→setUserData()network.set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

network→**set_userPassword()**
network→**setUserPassword()**
network.set_userPassword()

YNetwork

Changes the password for the "user" user.

```
function set_userPassword( newval: string): integer
```

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "user" user

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_wwwWatchdogDelay()**YNetwork****network→setWwwWatchdogDelay()****network.set_wwwWatchdogDelay()**

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
function set_wwwWatchdogDelay( newval: LongInt): integer
```

A zero value disables automated reboot in case of Internet connectivity loss. The smallest valid non-zero timeout is 90 seconds.

Parameters :

newval an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useDHCP()**network.useDHCP()****YNetwork**

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

```
function useDHCP( fallbackIpAddr: string,  
                  fallbackSubnetMaskLen: LongInt,  
                  fallbackRouter: string): integer
```

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

fallbackIpAddr	fallback IP address, to be used when no DHCP reply is received
fallbackSubnetMaskLen	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
fallbackRouter	fallback router IP address, to be used when no DHCP reply is received

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useStaticIP(network.useStaticIP())

YNetwork

Changes the configuration of the network interface to use a static IP address.

```
function useStaticIP( ipAddress: string,  
                    subnetMaskLen: LongInt,  
                    router: string): integer
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ipAddress device IP address
subnetMaskLen subnet mask length, as an integer (eg. 24 means 255.255.255.0)
router router IP address (default gateway)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.27. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_oscontrol.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YOsControl = yoctolib.YOsControl;
php	require_once('yocto_oscontrol.php');
c++	#include "yocto_oscontrol.h"
m	#import "yocto_oscontrol.h"
pas	uses yocto_oscontrol;
vb	yocto_oscontrol.vb
cs	yocto_oscontrol.cs
java	import com.yoctopuce.YoctoAPI.YOsControl;
py	from yocto_oscontrol import *

Global functions

yFindOsControl(func)

Retrieves OS control for a given identifier.

yFirstOsControl()

Starts the enumeration of OS control currently accessible.

YOsControl methods

oscontrol→describe()

Returns a short text that describes unambiguously the instance of the OS control in the form TYPE (NAME) = SERIAL . FUNCTIONID.

oscontrol→get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

oscontrol→get_errorMessage()

Returns the error message of the latest error with the OS control.

oscontrol→get_errorType()

Returns the numerical error code of the latest error with the OS control.

oscontrol→get_friendlyName()

Returns a global identifier of the OS control in the format MODULE_NAME . FUNCTION_NAME.

oscontrol→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

oscontrol→get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

oscontrol→get_hardwareId()

Returns the unique hardware identifier of the OS control in the form SERIAL . FUNCTIONID.

oscontrol→get_logicalName()

Returns the logical name of the OS control.

oscontrol→get_module()

Gets the YModule object for the device on which the function is located.

oscontrol→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`oscontrol→get_shutdownCountdown()`

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

`oscontrol→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

`oscontrol→isOnline()`

Checks if the OS control is currently reachable, without raising any error.

`oscontrol→isOnline_async(callback, context)`

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

`oscontrol→load(msValidity)`

Preloads the OS control cache with a specified validity duration.

`oscontrol→load_async(msValidity, callback, context)`

Preloads the OS control cache with a specified validity duration (asynchronous version).

`oscontrol→nextOsControl()`

Continues the enumeration of OS control started using `yFirstOsControl()`.

`oscontrol→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`oscontrol→set_logicalName(newval)`

Changes the logical name of the OS control.

`oscontrol→set_userData(data)`

Stores a user context provided as argument in the `userData` attribute of the function.

`oscontrol→shutdown(secBeforeShutDown)`

Schedules an OS shutdown after a given number of seconds.

`oscontrol→wait_async(callback, context)`

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YOsControl.FindOsControl() yFindOsControl()yFindOsControl()

YOsControl

Retrieves OS control for a given identifier.

```
function yFindOsControl( func: string): TYOsControl
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YOsControl.isOnline()` to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the OS control

Returns :

a `YOsControl` object allowing you to drive the OS control.

**YOsControl.FirstOsControl()
yFirstOsControl()yFirstOsControl()**

YOsControl

Starts the enumeration of OS control currently accessible.

```
function yFirstOsControl( ): TYOsControl
```

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

Returns :

a pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a `null` pointer if there are none.

oscontrol→describe()**oscontrol.describe()****YOsControl**

Returns a short text that describes unambiguously the instance of the OS control in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the OS control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

oscontrol→get_advertisedValue()**YOsControl****oscontrol→advertisedValue()****oscontrol.get_advertisedValue()**

Returns the current value of the OS control (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the OS control (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

oscontrol→get_errorMessage()

YOsControl

oscontrol→errorMessage()

oscontrol.get_errorMessage()

Returns the error message of the latest error with the OS control.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the OS control object

oscontrol→**get_errorType()****YOsControl****oscontrol**→**errorType()****oscontrol.get_errorType()**

Returns the numerical error code of the latest error with the OS control.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the OS control object

oscontrol→get_functionDescriptor()
oscontrol→functionDescriptor()
oscontrol.get_functionDescriptor()

YOsControl

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

oscontrol→get_logicalName()
oscontrol→logicalName()
oscontrol.get_logicalName()

YOsControl

Returns the logical name of the OS control.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the OS control. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

oscontrol→**get_module()**

YOsControl

oscontrol→**module()****oscontrol.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module()**: TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

oscontrol→get_shutdownCountdown()
oscontrol→shutdownCountdown()
oscontrol.get_shutdownCountdown()

YOsControl

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

```
function get_shutdownCountdown( ): LongInt
```

Returns :

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns Y_SHUTDOWNCOUNTDOWN_INVALID.

oscontrol→**get_userData()**

YOsControl

oscontrol→**userData()****oscontrol.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

oscontrol→isOnline()oscontrol.isOnline()**YOsControl**

Checks if the OS control is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

Returns :

`true` if the OS control can be reached, and `false` otherwise

oscontrol→load()oscontrol.load()**YOsControl**

Preloads the OS control cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

oscontrol→nextOsControl()
oscontrol.nextOsControl()

YOsControl

Continues the enumeration of OS control started using `yFirstOsControl()`.

```
function nextOsControl( ): TYOsControl
```

Returns :

a pointer to a `YOsControl` object, corresponding to OS control currently online, or a `null` pointer if there are no more OS control to enumerate.

oscontrol→registerValueCallback()
oscontrol.registerValueCallback()**YOsControl**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYOsControlValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

oscontrol→**set_logicalName()****YOsControl****oscontrol**→**setLogicalName()****oscontrol.set_logicalName()**

Changes the logical name of the OS control.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the OS control.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

oscontrol→**set_userdata()**

YOsControl

oscontrol→**setUserData()****oscontrol.set_userdata()**

Stores a user context provided as argument in the `userData` attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

oscontrol→shutdown()oscontrol.shutdown()

YOsControl

Schedules an OS shutdown after a given number of seconds.

```
function shutdown( secBeforeShutDown: LongInt): LongInt
```

Parameters :

secBeforeShutDown number of seconds before shutdown

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

3.28. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_power.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPower = yoctolib.YPower;
php	require_once('yocto_power.php');
c++	#include "yocto_power.h"
m	#import "yocto_power.h"
pas	uses yocto_power;
vb	yocto_power.vb
cs	yocto_power.cs
java	import com.yoctopuce.YoctoAPI.YPower;
py	from yocto_power import *

Global functions

yFindPower(func)

Retrieves a electrical power sensor for a given identifier.

yFirstPower()

Starts the enumeration of electrical power sensors currently accessible.

YPower methods

power→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

power→describe()

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form
TYPE (NAME) = SERIAL . FUNCTIONID.

power→get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

power→get_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

power→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

power→get_currentValue()

Returns the current measure for the electrical power.

power→get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

power→get_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

power→get_friendlyName()

Returns a global identifier of the electrical power sensor in the format
MODULE_NAME . FUNCTION_NAME.

power→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

power→**get_hardwareId()**

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

power→**get_highestValue()**

Returns the maximal value observed for the electrical power.

power→**get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→**get_logicalName()**

Returns the logical name of the electrical power sensor.

power→**get_lowestValue()**

Returns the minimal value observed for the electrical power.

power→**get_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

power→**get_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

power→**get_module()**

Gets the `YModule` object for the device on which the function is located.

power→**get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

power→**get_recordedData(startTime, endTime)**

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

power→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→**get_resolution()**

Returns the resolution of the measured values.

power→**get_unit()**

Returns the measuring unit for the electrical power.

power→**get_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

power→**isOnline()**

Checks if the electrical power sensor is currently reachable, without raising any error.

power→**isOnline_async(callback, context)**

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

power→**load(msValidity)**

Preloads the electrical power sensor cache with a specified validity duration.

power→**loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

power→**load_async(msValidity, callback, context)**

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

power→**nextPower()**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

power→**registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

power→**registerValueCallback(callback)**

3. Reference

	Registers the callback function that is invoked on every change of advertised value.
power→reset()	Resets the energy counter.
power→set_highestValue(newval)	Changes the recorded maximal value observed pour the electrical power.
power→set_logFrequency(newval)	Changes the datalogger recording frequency for this function.
power→set_logicalName(newval)	Changes the logical name of the electrical power sensor.
power→set_lowestValue(newval)	Changes the recorded minimal value observed pour the electrical power.
power→set_reportFrequency(newval)	Changes the timed value notification frequency for this function.
power→set_resolution(newval)	Changes the resolution of the measured values.
power→set_userData(data)	Stores a user context provided as argument in the userData attribute of the function.
power→wait_async(callback, context)	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPower.FindPower() yFindPower()yFindPower()

YPower

Retrieves a electrical power sensor for a given identifier.

```
function yFindPower( func: string): TYPower
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the electrical power sensor

Returns :

a `YPower` object allowing you to drive the electrical power sensor.

YPower.FirstPower() yFirstPower()yFirstPower()

YPower

Starts the enumeration of electrical power sensors currently accessible.

```
function yFirstPower( ): TYPower
```

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

Returns :

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

power→calibrateFromPoints()
power.calibrateFromPoints()

YPower

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→describe()power.describe()**YPower**

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the electrical power sensor (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

power→get_advertisedValue()**YPower****power→advertisedValue()****power.get_advertisedValue()**

Returns the current value of the electrical power sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the electrical power sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

power→**get_cosPhi()**

YPower

power→**cosPhi()****power.get_cosPhi()**

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

```
function get_cosPhi( ): double
```

Returns :

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns Y_COSPHI_INVALID.

power→**get_currentRawValue()****YPower****power**→**currentRawValue()****power.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

power→**get_currentValue()**

YPower

power→**currentValue()****power.get_currentValue()**

Returns the current measure for the electrical power.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current measure for the electrical power

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

power→**get_errorMessage()****YPower****power**→**errorMessage()****power.get_errorMessage()**

Returns the error message of the latest error with the electrical power sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the electrical power sensor object

power→**get_errorType()**

YPower

power→**errorType()****power.get_errorType()**

Returns the numerical error code of the latest error with the electrical power sensor.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the electrical power sensor object

power→**get_functionDescriptor()**
power→**functionDescriptor()**
power.get_functionDescriptor()

YPower

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

power→**get_highestValue()**

YPower

power→**highestValue()****power.get_highestValue()**

Returns the maximal value observed for the electrical power.

function **get_highestValue**(): double

Returns :

a floating point number corresponding to the maximal value observed for the electrical power

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

power→**get_logFrequency()****YPower****power**→**logFrequency()****power.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

power→**get_logicalName()**

YPower

power→**logicalName()****power.get_logicalName()**

Returns the logical name of the electrical power sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the electrical power sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

power→**get_lowestValue()****YPower****power**→**lowestValue()****power.get_lowestValue()**

Returns the minimal value observed for the electrical power.

function **get_lowestValue()**: double

Returns :

a floating point number corresponding to the minimal value observed for the electrical power

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

power→**get_meter()**

YPower

power→**meter()****power.get_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

```
function get_meter( ): double
```

Note that this counter is reset at each start of the device.

Returns :

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns `Y_METER_INVALID`.

power→**get_meterTimer()****YPower****power**→**meterTimer()****power.get_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

function **get_meterTimer**(): LongInt

Returns :

an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns Y_METERTIMER_INVALID.

power→**get_module()**

YPower

power→**module()****power.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module()**: TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

power→**get_recordedData()****YPower****power**→**recordedData()****power.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

power→get_reportFrequency()

YPower

power→reportFrequency()

power.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

power→**get_resolution()****YPower****power**→**resolution()****power.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

power→**get_unit()**

YPower

power→**unit()****power.get_unit()**

Returns the measuring unit for the electrical power.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns `Y_UNIT_INVALID`.

power→**get_userData()****YPower****power**→**userData()****power.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

power→isOnline()power.isOnline()

YPower

Checks if the electrical power sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

Returns :

`true` if the electrical power sensor can be reached, and `false` otherwise

power→load()power.load()**YPower**

Preloads the electrical power sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

power→loadCalibrationPoints()
power.loadCalibrationPoints()**YPower**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**nextPower()****power.nextPower()****YPower**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

```
function nextPower( ): TYPower
```

Returns :

a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a `null` pointer if there are no more electrical power sensors to enumerate.

power→registerTimedReportCallback()
power.registerTimedReportCallback()**YPower**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYPowerTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

power→registerValueCallback()
power.registerValueCallback()

YPower

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYPowerValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

power→reset()**power.reset()**

YPower

Resets the energy counter.

function **reset**(): LongInt

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_highestValue()****YPower****power**→**setHighestValue()****power.set_highestValue()**

Changes the recorded maximal value observed pour the electrical power.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed pour the electrical power

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_logFrequency()****YPower****power**→**setLogFrequency()****power.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_logicalName()****YPower****power**→**setLogicalName()****power.set_logicalName()**

Changes the logical name of the electrical power sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the electrical power sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

power→**set_lowestValue()****YPower****power**→**setLowestValue()****power.set_lowestValue()**

Changes the recorded minimal value observed pour the electrical power.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed pour the electrical power

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_reportFrequency()**
power→**setReportFrequency()**
power.set_reportFrequency()

YPower

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_resolution()****YPower****power**→**setResolution()****power.set_resolution()**

Changes the resolution of the measured values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→**set_userData()****YPower****power**→**setUserData()****power.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userData( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.29. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pressure.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPressure = yoctolib.YPressure;
php	require_once('yocto_pressure.php');
c++	#include "yocto_pressure.h"
m	#import "yocto_pressure.h"
pas	uses yocto_pressure;
vb	yocto_pressure.vb
cs	yocto_pressure.cs
java	import com.yoctopuce.YoctoAPI.YPressure;
py	from yocto_pressure import *

Global functions

yFindPressure(func)

Retrieves a pressure sensor for a given identifier.

yFirstPressure()

Starts the enumeration of pressure sensors currently accessible.

YPressure methods

pressure→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

pressure→describe()

Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

pressure→get_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

pressure→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

pressure→get_currentValue()

Returns the current measure for the pressure.

pressure→get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

pressure→get_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

pressure→get_friendlyName()

Returns a global identifier of the pressure sensor in the format `MODULE_NAME . FUNCTION_NAME`.

pressure→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pressure→get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

pressure→get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL . FUNCTIONID`.

pressure→get_highestValue()

Returns the maximal value observed for the pressure.

pressure→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pressure→get_logicalName()

Returns the logical name of the pressure sensor.

pressure→get_lowestValue()

Returns the minimal value observed for the pressure.

pressure→get_module()

Gets the `YModule` object for the device on which the function is located.

pressure→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pressure→get_recordedData(startTime, endTime)

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

pressure→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pressure→get_resolution()

Returns the resolution of the measured values.

pressure→get_unit()

Returns the measuring unit for the pressure.

pressure→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

pressure→isOnline()

Checks if the pressure sensor is currently reachable, without raising any error.

pressure→isOnline_async(callback, context)

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

pressure→load(msValidity)

Preloads the pressure sensor cache with a specified validity duration.

pressure→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

pressure→load_async(msValidity, callback, context)

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

pressure→nextPressure()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

pressure→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

pressure→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pressure→set_highestValue(newval)

Changes the recorded maximal value observed for the pressure.

pressure→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

pressure→set_logicalName(newval)

Changes the logical name of the pressure sensor.

3. Reference

pressure→**set_lowestValue**(newval)

Changes the recorded minimal value observed for the pressure.

pressure→**set_reportFrequency**(newval)

Changes the timed value notification frequency for this function.

pressure→**set_resolution**(newval)

Changes the resolution of the measured physical values.

pressure→**set_userData**(data)

Stores a user context provided as argument in the userData attribute of the function.

pressure→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPressure.FindPressure() yFindPressure()yFindPressure()

YPressure

Retrieves a pressure sensor for a given identifier.

```
function yFindPressure( func: string): TYPresure
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the pressure sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPressure.isOnline()` to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the pressure sensor

Returns :

a `YPressure` object allowing you to drive the pressure sensor.

YPressure.FirstPressure() yFirstPressure()yFirstPressure()

YPressure

Starts the enumeration of pressure sensors currently accessible.

```
function yFirstPressure( ): TYPPressure
```

Use the method `YPressure.nextPressure()` to iterate on next pressure sensors.

Returns :

a pointer to a `YPressure` object, corresponding to the first pressure sensor currently online, or a `null` pointer if there are none.

pressure→calibrateFromPoints()
pressure.calibrateFromPoints()

YPressure

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                               refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→describe()pressure.describe()**YPressure**

Returns a short text that describes unambiguously the instance of the pressure sensor in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the pressure sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pressure→**get_advertisedValue()****YPressure****pressure**→**advertisedValue()****pressure.get_advertisedValue()**

Returns the current value of the pressure sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the pressure sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pressure→**get_currentRawValue()**

YPressure

pressure→**currentRawValue()**

pressure.get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

pressure→**get_currentValue()****YPressure****pressure**→**currentValue()****pressure.get_currentValue()**

Returns the current measure for the pressure.

function **get_currentValue**(): double

Returns :

a floating point number corresponding to the current measure for the pressure

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

pressure→**get_errorMessage()**

YPressure

pressure→**errorMessage()**

pressure.get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the pressure sensor object

pressure→**get_errorType()****YPressure****pressure**→**errorType()****pressure.get_errorType()**

Returns the numerical error code of the latest error with the pressure sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the pressure sensor object

pressure→**get_functionDescriptor()**
pressure→**functionDescriptor()**
pressure.get_functionDescriptor()

YPressure

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pressure→**get_highestValue()****YPressure****pressure**→**highestValue()****pressure.get_highestValue()**

Returns the maximal value observed for the pressure.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the pressure

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

pressure→**get_logFrequency()**

YPressure

pressure→**logFrequency()**

pressure.get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

pressure→**get_logicalName()****YPressure****pressure**→**logicalName()****pressure.get_logicalName()**

Returns the logical name of the pressure sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the pressure sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pressure→**get_lowestValue()**

YPressure

pressure→**lowestValue()****pressure.get_lowestValue()**

Returns the minimal value observed for the pressure.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the pressure

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

pressure→**get_module()****YPressure****pressure**→**module()****pressure.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

pressure→**get_recordedData()**
pressure→**recordedData()**
pressure.get_recordedData()

YPressure

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pressure→get_reportFrequency()**YPressure****pressure→reportFrequency()****pressure.get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

pressure→**get_resolution()**

YPressure

pressure→**resolution()****pressure.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

pressure→**get_unit()****YPressure****pressure**→**unit()****pressure.get_unit()**

Returns the measuring unit for the pressure.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns Y_UNIT_INVALID.

pressure→**get_userData()**

YPressure

pressure→**userData()****pressure.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

pressure→**isOnline()****pressure.isOnline()****YPressure**

Checks if the pressure sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

Returns :

`true` if the pressure sensor can be reached, and `false` otherwise

pressure→**load()****pressure.load()****YPressure**

Preloads the pressure sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pressure→loadCalibrationPoints()
pressure.loadCalibrationPoints()

YPressure

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**nextPressure()****pressure.nextPressure()**

YPressure

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

```
function nextPressure( ): TYPressure
```

Returns :

a pointer to a YPressure object, corresponding to a pressure sensor currently online, or a `null` pointer if there are no more pressure sensors to enumerate.

pressure→registerTimedReportCallback()
pressure.registerTimedReportCallback()

YPressure

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYPresureTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

pressure→registerValueCallback()
pressure.registerValueCallback()**YPressure**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYPresureValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pressure→**set_highestValue()****YPressure****pressure**→**setHighestValue()****pressure.set_highestValue()**

Changes the recorded maximal value observed for the pressure.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the pressure

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**set_logFrequency()****YPressure****pressure**→**setLogFrequency()****pressure.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**set_logicalName()**
pressure→**setLogicalName()**
pressure.set_logicalName()

YPressure

Changes the logical name of the pressure sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the pressure sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

pressure→**set_lowestValue()**

YPressure

pressure→**setLowestValue()**

pressure.set_lowestValue()

Changes the recorded minimal value observed for the pressure.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the pressure

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_reportFrequency()
pressure→setReportFrequency()
pressure.set_reportFrequency()

YPressure

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**set_resolution()****YPressure****pressure**→**setResolution()****pressure.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→**set_userData()****YPressure****pressure**→**setUserData()****pressure.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userData( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.30. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmoutput.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPwmOutput = yoctolib.YPwmOutput;
php	require_once('yocto_pwmoutput.php');
c++	#include "yocto_pwmoutput.h"
m	#import "yocto_pwmoutput.h"
pas	uses yocto_pwmoutput;
vb	yocto_pwmoutput.vb
cs	yocto_pwmoutput.cs
java	import com.yoctopuce.YoctoAPI.YPwmOutput;
py	from yocto_pwmoutput import *

Global functions

yFindPwmOutput(func)

Retrieves a PWM for a given identifier.

yFirstPwmOutput()

Starts the enumeration of PWMs currently accessible.

YPwmOutput methods

pwmoutput→describe()

Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

pwmoutput→dutyCycleMove(target, ms_duration)

Performs a smooth change of the pulse duration toward a given value.

pwmoutput→get_advertisedValue()

Returns the current value of the PWM (no more than 6 characters).

pwmoutput→get_dutyCycle()

Returns the PWM duty cycle, in per cents.

pwmoutput→get_dutyCycleAtPowerOn()

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

pwmoutput→get_enabled()

Returns the state of the PWMs.

pwmoutput→get_enabledAtPowerOn()

Returns the state of the PWM at device power on.

pwmoutput→get_errorMessage()

Returns the error message of the latest error with the PWM.

pwmoutput→get_errorType()

Returns the numerical error code of the latest error with the PWM.

pwmoutput→get_frequency()

Returns the PWM frequency in Hz.

pwmoutput→get_friendlyName()

Returns a global identifier of the PWM in the format `MODULE_NAME . FUNCTION_NAME`.

pwmoutput→get_functionDescriptor()

Returns a unique identifier of type `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.

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.

3. Reference

pwmoutput→set_pulseDuration(newval)

Changes the PWM pulse length, in milliseconds.

pwmoutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmoutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmOutput.FindPwmOutput() yFindPwmOutput()yFindPwmOutput()

YPwmOutput

Retrieves a PWM for a given identifier.

```
function yFindPwmOutput( func: string): TYPwmOutput
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the PWM

Returns :

a `YPwmOutput` object allowing you to drive the PWM.

YPwmOutput.FirstPwmOutput() yFirstPwmOutput()yFirstPwmOutput()

YPwmOutput

Starts the enumeration of PWMs currently accessible.

```
function yFirstPwmOutput( ): TYPwmOutput
```

Use the method `YPwmOutput.nextPwmOutput()` to iterate on next PWMs.

Returns :

a pointer to a `YPwmOutput` object, corresponding to the first PWM currently online, or a `null` pointer if there are none.

pwmoutput→describe()**pwmoutput.describe()****YPwmOutput**

Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the PWM (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmoutput→dutyCycleMove()
pwmoutput.dutyCycleMove()**YPwmOutput**

Performs a smooth change of the pulse duration toward a given value.

```
function dutyCycleMove( target: double, ms_duration: LongInt): LongInt
```

Parameters :

target new duty cycle at the end of the transition (floating-point number, between 0 and 1)
ms_duration total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→get_advertisedValue()**YPwmOutput****pwmoutput→advertisedValue()****pwmoutput.get_advertisedValue()**

Returns the current value of the PWM (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the PWM (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwmoutput→get_dutyCycle()

YPwmOutput

pwmoutput→dutyCycle()**pwmoutput.get_dutyCycle()**

Returns the PWM duty cycle, in per cents.

```
function get_dutyCycle( ): double
```

Returns :

a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns Y_DUTYCYCLE_INVALID.

pwmoutput→get_dutyCycleAtPowerOn()
pwmoutput→dutyCycleAtPowerOn()
pwmoutput.get_dutyCycleAtPowerOn()

YPwmOutput

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

```
function get_dutyCycleAtPowerOn( ): double
```

Returns :

a floating point number corresponding to the PWMs duty cycle at device power on as a floating point number between 0 and 100

On failure, throws an exception or returns Y_DUTYCYCLEATPOWERON_INVALID.

pwmoutput→get_enabled()

YPwmOutput

pwmoutput→enabled()**pwmoutput.get_enabled()**

Returns the state of the PWMs.

function **get_enabled**(): Integer

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the PWMs

On failure, throws an exception or returns Y_ENABLED_INVALID.

pwmoutput→get_enabledAtPowerOn()
pwmoutput→enabledAtPowerOn()
pwmoutput.get_enabledAtPowerOn()

YPwmOutput

Returns the state of the PWM at device power on.

```
function get_enabledAtPowerOn( ): Integer
```

Returns :

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

pwmoutput→get_errorMessage()
pwmoutput→errorMessage()
pwmoutput.get_errorMessage()

YPwmOutput

Returns the error message of the latest error with the PWM.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the PWM object

pwmoutput→get_errorType()**YPwmOutput****pwmoutput→errorType()pwmoutput.get_errorType()**

Returns the numerical error code of the latest error with the PWM.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the PWM object

pwmoutput→get_frequency()

YPwmOutput

pwmoutput→frequency()**pwmoutput.get_frequency()**

Returns the PWM frequency in Hz.

function **get_frequency()**: LongInt

Returns :

an integer corresponding to the PWM frequency in Hz

On failure, throws an exception or returns Y_FREQUENCY_INVALID.

pwmoutput→get_functionDescriptor()
pwmoutput→functionDescriptor()
pwmoutput.get_functionDescriptor()

YPwmOutput

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmoutput→get_logicalName()

YPwmOutput

pwmoutput→logicalName()

pwmoutput.get_logicalName()

Returns the logical name of the PWM.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the PWM. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pwmoutput→get_module()**YPwmOutput****pwmoutput→module()pwmoutput.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

pwmoutput→get_period()

YPwmOutput

pwmoutput→period()pwmoutput.get_period()

Returns the PWM period in milliseconds.

```
function get_period( ): double
```

Returns :

a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns Y_PERIOD_INVALID.

pwmoutput→get_pulseDuration()**YPwmOutput****pwmoutput→pulseDuration()****pwmoutput.get_pulseDuration()**

Returns the PWM pulse length in milliseconds.

```
function get_pulseDuration( ): double
```

Returns :

a floating point number corresponding to the PWM pulse length in milliseconds

On failure, throws an exception or returns Y_PULSEDURATION_INVALID.

pwmoutput→get_userdata()

YPwmOutput

pwmoutput→userdata()pwmoutput.get_userdata()

Returns the value of the `userData` attribute, as previously stored using method `set_userdata`.

```
function get_userdata( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

pwmoutput→isOnline()pwmoutput.isOnline()

YPwmOutput

Checks if the PWM is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

Returns :

`true` if the PWM can be reached, and `false` otherwise

pwmoutput→load()**pwmoutput.load()****YPwmOutput**

Preloads the PWM cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→nextPwmOutput()
pwmoutput.nextPwmOutput()**YPwmOutput**

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

```
function nextPwmOutput( ): TYPwmOutput
```

Returns :

a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.

**pwmoutput→pulseDurationMove()
pwmoutput.pulseDurationMove()****YPwmOutput**

Performs a smooth transistion of the pulse duration toward a given value.

```
function pulseDurationMove( ms_target: double,  
                             ms_duration: LongInt): LongInt
```

Any period, frequency, duty cycle or pulse width change will cancel any ongoing transition process.

Parameters :

ms_target new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)
ms_duration total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→registerValueCallback()
pwmoutput.registerValueCallback()

YPwmOutput

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYPwmOutputValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmoutput→set_dutyCycle()

YPwmOutput

pwmoutput→setDutyCycle()

pwmoutput.set_dutyCycle()

Changes the PWM duty cycle, in per cents.

```
function set_dutyCycle( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the PWM duty cycle, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_dutyCycleAtPowerOn()
pwmoutput→setDutyCycleAtPowerOn()
pwmoutput.set_dutyCycleAtPowerOn()

YPwmOutput

Changes the PWM duty cycle at device power on.

```
function set_dutyCycleAtPowerOn( newval: double): integer
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a floating point number corresponding to the PWM duty cycle at device power on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabled()

YPwmOutput

pwmoutput→setEnabled()**pwmoutput.set_enabled()**

Stops or starts the PWM.

```
function set_enabled( newval: Integer): integer
```

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabledAtPowerOn()
pwmoutput→setEnabledAtPowerOn()
pwmoutput.set_enabledAtPowerOn()

YPwmOutput

Changes the state of the PWM at device power on.

```
function set_enabledAtPowerOn( newval: Integer): integer
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_frequency()
pwmoutput→setFrequency()
pwmoutput.set_frequency()

YPwmOutput

Changes the PWM frequency.

```
function set_frequency( newval: LongInt): integer
```

The duty cycle is kept unchanged thanks to an automatic pulse width change.

Parameters :

newval an integer corresponding to the PWM frequency

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_logicalName()**YPwmOutput****pwmoutput→setLogicalName()****pwmoutput.set_logicalName()**

Changes the logical name of the PWM.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the PWM.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→set_period()

YPwmOutput

pwmoutput→setPeriod()**pwmoutput.set_period()**

Changes the PWM period.

```
function set_period( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the PWM period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_pulseDuration()
pwmoutput→setPulseDuration()
pwmoutput.set_pulseDuration()

YPwmOutput

Changes the PWM pulse length, in milliseconds.

```
function set_pulseDuration( newval: double): integer
```

A pulse length cannot be longer than period, otherwise it is truncated.

Parameters :

newval a floating point number corresponding to the PWM pulse length, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_userdata()
pwmoutput→setUserData()
pwmoutput.set_userdata()

YPwmOutput

Stores a user context provided as argument in the `userData` attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.31. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmpowersource.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPwmPowerSource = yoctolib.YPwmPowerSource;
php	require_once('yocto_pwmpowersource.php');
c++	#include "yocto_pwmpowersource.h"
m	#import "yocto_pwmpowersource.h"
pas	uses yocto_pwmpowersource;
vb	yocto_pwmpowersource.vb
cs	yocto_pwmpowersource.cs
java	import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py	from yocto_pwmpowersource import *

Global functions

yFindPwmPowerSource(func)

Retrieves a voltage source for a given identifier.

yFirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

YPwmPowerSource methods

pwmpowersource→describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form TYPE (NAME) = SERIAL . FUNCTIONID.

pwmpowersource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

pwmpowersource→get_errorMessage()

Returns the error message of the latest error with the voltage source.

pwmpowersource→get_errorType()

Returns the numerical error code of the latest error with the voltage source.

pwmpowersource→get_friendlyName()

Returns a global identifier of the voltage source in the format MODULE_NAME . FUNCTION_NAME.

pwmpowersource→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

pwmpowersource→get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

pwmpowersource→get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form SERIAL . FUNCTIONID.

pwmpowersource→get_logicalName()

Returns the logical name of the voltage source.

pwmpowersource→get_module()

Gets the YModule object for the device on which the function is located.

pwmpowersource→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

pwmpowersource→get_powerMode()

Returns the selected power source for the PWM on the same device

pwmpowersource→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

pwmpowersource→isOnline()

Checks if the voltage source is currently reachable, without raising any error.

pwmpowersource→isOnline_async(callback, context)

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

pwmpowersource→load(msValidity)

Preloads the voltage source cache with a specified validity duration.

pwmpowersource→load_async(msValidity, callback, context)

Preloads the voltage source cache with a specified validity duration (asynchronous version).

pwmpowersource→nextPwmPowerSource()

Continues the enumeration of Voltage sources started using yFirstPwmPowerSource().

pwmpowersource→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pwmpowersource→set_logicalName(newval)

Changes the logical name of the voltage source.

pwmpowersource→set_powerMode(newval)

Changes the PWM power source.

pwmpowersource→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmpowersource→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPwmPowerSource.FindPwmPowerSource() yFindPwmPowerSource()yFindPwmPowerSource()

YPwmPowerSource

Retrieves a voltage source for a given identifier.

```
function yFindPwmPowerSource( func: string): TYPwmPowerSource
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmPowerSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage source

Returns :

a `YPwmPowerSource` object allowing you to drive the voltage source.

**YPwmPowerSource.FirstPwmPowerSource()
yFirstPwmPowerSource()yFirstPwmPowerSource()**

YPwmPowerSource

Starts the enumeration of Voltage sources currently accessible.

```
function yFirstPwmPowerSource( ): TYPwmPowerSource
```

Use the method `YPwmPowerSource.nextPwmPowerSource()` to iterate on next Voltage sources.

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to the first source currently online, or a `null` pointer if there are none.

pwmpowersource→describe()
pwmpowersource.describe()

YPwmPowerSource

Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage source (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmpowersource→get_advertisedValue()

YPwmPowerSource

pwmpowersource→advertisedValue()

pwmpowersource.get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the voltage source (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

pwmpowersource→get_errorMessage()
pwmpowersource→errorMessage()
pwmpowersource.get_errorMessage()

YPwmPowerSource

Returns the error message of the latest error with the voltage source.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the voltage source object

pwmpowersource→get_errorType()

YPwmPowerSource

pwmpowersource→errorType()

pwmpowersource.get_errorType()

Returns the numerical error code of the latest error with the voltage source.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the voltage source object

pwmpowersource→get_functionDescriptor()
pwmpowersource→functionDescriptor()
pwmpowersource.get_functionDescriptor()

YPwmPowerSource

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

pwmpowersource→get_logicalName()

YPwmPowerSource

pwmpowersource→logicalName()

pwmpowersource.get_logicalName()

Returns the logical name of the voltage source.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the voltage source. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

pwmpowersource→get_module()
pwmpowersource→module()
pwmpowersource.get_module()

YPwmPowerSource

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

pwmpowersource→get_powerMode()

YPwmPowerSource

pwmpowersource→powerMode()

pwmpowersource.get_powerMode()

Returns the selected power source for the PWM on the same device

function **get_powerMode**(): Integer

Returns :

a value among Y_POWERMODE_USB_5V, Y_POWERMODE_USB_3V, Y_POWERMODE_EXT_V and Y_POWERMODE_OPNDRN corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns Y_POWERMODE_INVALID.

pwmpowersource→get_userData()
pwmpowersource→userData()
pwmpowersource.get_userData()

YPwmPowerSource

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

pwmpowersource→**isOnline()**
pwmpowersource.isOnline()

YPwmPowerSource

Checks if the voltage source is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

Returns :

`true` if the voltage source can be reached, and `false` otherwise

pwmpowersource→load()pwmpowersource.load()

YPwmPowerSource

Preloads the voltage source cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmpowersource→nextPwmPowerSource()
pwmpowersource.nextPwmPowerSource()

YPwmPowerSource

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

```
function nextPwmPowerSource( ): TYPwmPowerSource
```

Returns :

a pointer to a `YPwmPowerSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more Voltage sources to enumerate.

pwmpowersource→registerValueCallback()
pwmpowersource.registerValueCallback()

YPwmPowerSource

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYPwmPowerSourceValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmpowersource→set_logicalName()
pwmpowersource→setLogicalName()
pwmpowersource.set_logicalName()

YPwmPowerSource

Changes the logical name of the voltage source.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage source.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

pwmpowersource→set_powerMode()
pwmpowersource→setPowerMode()
pwmpowersource.set_powerMode()

YPwmPowerSource

Changes the PWM power source.

```
function set_powerMode( newval: Integer): integer
```

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

Parameters :

newval a value among Y_POWERMODE_USB_5V, Y_POWERMODE_USB_3V, Y_POWERMODE_EXT_V and Y_POWERMODE_OPNDRN corresponding to the PWM power source

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→set_userdata()

YPwmPowerSource

pwmpowersource→setUserData()

pwmpowersource.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.32. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_gyro.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib'); var YGyro = yoctolib.YGyro;</code>
php	<code>require_once('yocto_gyro.php');</code>
c++	<code>#include "yocto_gyro.h"</code>
m	<code>#import "yocto_gyro.h"</code>
pas	<code>uses yocto_gyro;</code>
vb	<code>yocto_gyro.vb</code>
cs	<code>yocto_gyro.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YGyro;</code>
py	<code>from yocto_gyro import *</code>

Global functions

yFindQt(func)

Retrieves a quaternion component for a given identifier.

yFirstQt()

Starts the enumeration of quaternion components currently accessible.

YQt methods

qt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

qt→describe()

Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

qt→get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

qt→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

qt→get_currentValue()

Returns the current value of the value.

qt→get_errorMessage()

Returns the error message of the latest error with the quaternion component.

qt→get_errorType()

Returns the numerical error code of the latest error with the quaternion component.

qt→get_friendlyName()

Returns a global identifier of the quaternion component in the format `MODULE_NAME . FUNCTION_NAME`.

qt→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

qt→get_functionId()

Returns the hardware identifier of the quaternion component, without reference to the module.

qt→get_hardwareId()

	Returns the unique hardware identifier of the quaternion component in the form <code>SERIAL.FUNCTIONID</code> .
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 <code>YModule</code> object for the device on which the function is located.
qt→get_module_async(callback, context)	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
qt→get_recordedData(startTime, endTime)	Retrieves a <code>DataSet</code> 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 <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
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 <code>calibrateFromPoints</code> .
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 <code>yFirstQt()</code> .
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.

YQt.FindQt() yFindQt(yFindQt())

YQt

Retrieves a quaternion component for a given identifier.

```
function yFindQt( func: string): TYQt
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQt.isOnline()` to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the quaternion component

Returns :

a `YQt` object allowing you to drive the quaternion component.

YQt.FirstQt()**YQt****yFirstQt()**

Starts the enumeration of quaternion components currently accessible.

```
function yFirstQt( ): TYQt
```

Use the method `YQt.nextQt()` to iterate on next quaternion components.

Returns :

a pointer to a `YQt` object, corresponding to the first quaternion component currently online, or a `null` pointer if there are none.

qt→calibrateFromPoints()qt.calibrateFromPoints()**YQt**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→describe()qt.describe()**YQt**

Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the quaternion component (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

qt→get_advertisedValue()

YQt

qt→advertisedValue()qt.get_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the quaternion component (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

qt→get_currentRawValue()**YQt****qt→currentRawValue()qt.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

qt→get_currentValue()

YQt

qt→currentValue()qt.get_currentValue()

Returns the current value of the value.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

qt→get_errorMessage()**YQt****qt→errorMessage()qt.get_errorMessage()**

Returns the error message of the latest error with the quaternion component.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the quaternion component object

qt→get_errorType()

YQt

qt→errorType()qt.get_errorType()

Returns the numerical error code of the latest error with the quaternion component.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the quaternion component object

qt→get_functionDescriptor()**YQt****qt→functionDescriptor()qt.get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

qt→get_highestValue()

YQt

qt→highestValue()qt.get_highestValue()

Returns the maximal value observed for the value since the device was started.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

qt→get_logFrequency()**YQt****qt→logFrequency()qt.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

qt→get_logicalName()

YQt

qt→logicalName()qt.get_logicalName()

Returns the logical name of the quaternion component.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the quaternion component. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

qt→get_lowestValue()**YQt****qt→lowestValue()qt.get_lowestValue()**

Returns the minimal value observed for the value since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the value since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

qt→get_module()

YQt

qt→module()qt.get_module()

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TYModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

qt→get_recordedData()**YQt****qt→recordedData()qt.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

qt→get_reportFrequency()

YQt

qt→reportFrequency()qt.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

qt→get_resolution()**YQt****qt→resolution()qt.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

qt→get_unit()

YQt

qt→unit()qt.get_unit()

Returns the measuring unit for the value.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the value

On failure, throws an exception or returns Y_UNIT_INVALID.

qt→get_userdata()**YQt****qt→userdata()qt.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

function **get_userdata()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

qt→isOnline()qt.isOnline()**YQt**

Checks if the quaternion component is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

Returns :

`true` if the quaternion component can be reached, and `false` otherwise

qt→load()qt.load()**YQt**

Preloads the quaternion component cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

qt→loadCalibrationPoints()qt.loadCalibrationPoints()**YQt**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→nextQt()qt.nextQt()**YQt**

Continues the enumeration of quaternion components started using `yFirstQt()`.

```
function nextQt( ): TYQt
```

Returns :

a pointer to a `YQt` object, corresponding to a quaternion component currently online, or a `null` pointer if there are no more quaternion components to enumerate.

qt→registerTimedReportCallback()
qt.registerTimedReportCallback()

YQt

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYQtTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**qt→registerValueCallback()
qt.registerValueCallback()****YQt**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYQtValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

qt→set_highestValue()**YQt****qt→setHighestValue()qt.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logFrequency()**YQt****qt→setLogFrequency()qt.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→**set_logicalName()****YQt****qt**→**setLogicalName()****qt.set_logicalName()**

Changes the logical name of the quaternion component.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the quaternion component.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

qt→**set_lowestValue()****YQt****qt**→**setLowestValue()****qt.set_lowestValue()**

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_reportFrequency()**YQt****qt→setReportFrequency()qt.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_resolution()**YQt****qt→setResolution()qt.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_userdata()

YQt

qt→setUserData()qt.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.33. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_realtimeclock.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YRealTimeClock = yoctolib.YRealTimeClock;
php	require_once('yocto_realtimeclock.php');
c++	#include "yocto_realtimeclock.h"
m	#import "yocto_realtimeclock.h"
pas	uses yocto_realtimeclock;
vb	yocto_realtimeclock.vb
cs	yocto_realtimeclock.cs
java	import com.yoctopuce.YoctoAPI.YRealTimeClock;
py	from yocto_realtimeclock import *

Global functions

yFindRealTimeClock(func)

Retrieves a clock for a given identifier.

yFirstRealTimeClock()

Starts the enumeration of clocks currently accessible.

YRealTimeClock methods

realtimeclock→describe()

Returns a short text that describes unambiguously the instance of the clock in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

realtimeclock→get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

realtimeclock→get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

realtimeclock→get_errorMessage()

Returns the error message of the latest error with the clock.

realtimeclock→get_errorType()

Returns the numerical error code of the latest error with the clock.

realtimeclock→get_friendlyName()

Returns a global identifier of the clock in the format `MODULE_NAME . FUNCTION_NAME`.

realtimeclock→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

realtimeclock→get_functionId()

Returns the hardware identifier of the clock, without reference to the module.

realtimeclock→get_hardwareId()

Returns the unique hardware identifier of the clock in the form `SERIAL . FUNCTIONID`.

realtimeclock→get_logicalName()

Returns the logical name of the clock.

realtimeclock→get_module()

	Gets the <code>YModule</code> object for the device on which the function is located.
<code>realtimeclock→get_module_async(callback, context)</code>	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<code>realtimeclock→get_timeSet()</code>	Returns true if the clock has been set, and false otherwise.
<code>realtimeclock→get_unixTime()</code>	Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).
<code>realtimeclock→get_userData()</code>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<code>realtimeclock→get_utcOffset()</code>	Returns the number of seconds between current time and UTC time (time zone).
<code>realtimeclock→isOnline()</code>	Checks if the clock is currently reachable, without raising any error.
<code>realtimeclock→isOnline_async(callback, context)</code>	Checks if the clock is currently reachable, without raising any error (asynchronous version).
<code>realtimeclock→load(msValidity)</code>	Preloads the clock cache with a specified validity duration.
<code>realtimeclock→load_async(msValidity, callback, context)</code>	Preloads the clock cache with a specified validity duration (asynchronous version).
<code>realtimeclock→nextRealTimeClock()</code>	Continues the enumeration of clocks started using <code>yFirstRealTimeClock()</code> .
<code>realtimeclock→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>realtimeclock→set_logicalName(newval)</code>	Changes the logical name of the clock.
<code>realtimeclock→set_unixTime(newval)</code>	Changes the current time.
<code>realtimeclock→set_userData(data)</code>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<code>realtimeclock→set_utcOffset(newval)</code>	Changes the number of seconds between current time and UTC time (time zone).
<code>realtimeclock→wait_async(callback, context)</code>	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRealTimeClock.FindRealTimeClock() yFindRealTimeClock()yFindRealTimeClock()

YRealTimeClock

Retrieves a clock for a given identifier.

```
function yFindRealTimeClock( func: string): TYRealTimeClock
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the clock is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRealTimeClock.isOnline()` to test if the clock is indeed online at a given time. In case of ambiguity when looking for a clock by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the clock

Returns :

a `YRealTimeClock` object allowing you to drive the clock.

YRealTimeClock.FirstRealTimeClock() yFirstRealTimeClock()

YRealTimeClock

Starts the enumeration of clocks currently accessible.

```
function yFirstRealTimeClock( ): TYRealTimeClock
```

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

Returns :

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a `null` pointer if there are none.

realtimeclock→describe()realtimeclock.describe()**YRealTimeClock**

Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the clock (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

realtimeclock→get_advertisedValue()

YRealTimeClock

realtimeclock→advertisedValue()

realtimeclock.get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the clock (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

realtimeclock→get_dateTime()**YRealTimeClock****realtimeclock→dateTime()****realtimeclock.get_dateTime()**

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

```
function get_dateTime( ): string
```

Returns :

a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns Y_DATETIME_INVALID.

realtimeclock→get_errorMessage()
realtimeclock→errorMessage()
realtimeclock.get_errorMessage()

YRealTimeClock

Returns the error message of the latest error with the clock.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the clock object

realtimeclock→get_errorType()
realtimeclock→errorType()
realtimeclock.get_errorType()

YRealTimeClock

Returns the numerical error code of the latest error with the clock.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the clock object

realtimeclock→get_functionDescriptor()
realtimeclock→functionDescriptor()
realtimeclock.get_functionDescriptor()

YRealTimeClock

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

realtimeclock→get_logicalName()**YRealTimeClock****realtimeclock→logicalName()****realtimeclock.get_logicalName()**

Returns the logical name of the clock.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the clock. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

realtimeclock→get_module()

YRealTimeClock

realtimeclock→module()realtimeclock.get_module()

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TYModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

realtimeclock→**get_timeSet()****YRealTimeClock****realtimeclock**→**timeSet()****realtimeclock.get_timeSet()**

Returns true if the clock has been set, and false otherwise.

```
function get_timeSet( ): Integer
```

Returns :

either Y_TIMESET_FALSE or Y_TIMESET_TRUE, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns Y_TIMESET_INVALID.

realtimeclock→get_unixTime()
realtimeclock→unixTime()
realtimeclock.get_unixTime()

YRealTimeClock

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

function **get_unixTime**(): int64

Returns :

an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns Y_UNIXTIME_INVALID.

realtimeclock→get_userdata()
realtimeclock→userData()
realtimeclock.get_userdata()

YRealTimeClock

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
function get_userdata( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

realtimeclock→get_utcOffset()

YRealTimeClock

realtimeclock→utcOffset()

realtimeclock.get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

function **get_utcOffset**(): LongInt

Returns :

an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns Y_UTC_OFFSET_INVALID.

realtimeclock→**isOnline()****realtimeclock.isOnline()****YRealTimeClock**

Checks if the clock is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

Returns :

`true` if the clock can be reached, and `false` otherwise

realtimeclock→load()realtimeclock.load()**YRealTimeClock**

Preloads the clock cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

realtimeclock→nextRealTimeClock()
realtimeclock.nextRealTimeClock()

YRealTimeClock

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

```
function nextRealTimeClock( ): TYRealTimeClock
```

Returns :

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a `null` pointer if there are no more clocks to enumerate.

realtimeclock→registerValueCallback()
realtimeclock.registerValueCallback()**YRealTimeClock**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYRealTimeClockValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

realtimeclock→set_logicalName()
realtimeclock→setLogicalName()
realtimeclock.set_logicalName()

YRealTimeClock

Changes the logical name of the clock.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the clock.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

realtimeclock→set_unixTime()
realtimeclock→setUnixTime()
realtimeclock.set_unixTime()

YRealTimeClock

Changes the current time.

```
function set_unixTime( newval: int64): integer
```

Time is specifid in Unix format (number of elapsed seconds since Jan 1st, 1970). If current UTC time is known, utcOffset will be automatically adjusted for the new specified time.

Parameters :

newval an integer corresponding to the current time

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→set_userdata()
realtimeclock→setUserData()
realtimeclock.set_userdata()

YRealTimeClock

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

realtimeclock→set_utcOffset()

YRealTimeClock

realtimeclock→setUtcOffset()

realtimeclock.set_utcOffset()

Changes the number of seconds between current time and UTC time (time zone).

```
function set_utcOffset( newval: LongInt): integer
```

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time will automatically be updated according to the selected time zone.

Parameters :

newval an integer corresponding to the number of seconds between current time and UTC time (time zone)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.34. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_reframe.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YRefFrame = yoctolib.YRefFrame;
php	require_once('yocto_reframe.php');
c++	#include "yocto_reframe.h"
m	#import "yocto_reframe.h"
pas	uses yocto_reframe;
vb	yocto_reframe.vb
cs	yocto_reframe.cs
java	import com.yoctopuce.YoctoAPI.YRefFrame;
py	from yocto_reframe import *

Global functions

yFindRefFrame(func)

Retrieves a reference frame for a given identifier.

yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

YRefFrame methods

reframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

reframe→describe()

Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

reframe→get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

reframe→get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

reframe→get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

reframe→get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

reframe→get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

reframe→get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

reframe→get_bearing()

Returns the reference bearing used by the compass.

reiframe→get_errorMessage()

Returns the error message of the latest error with the reference frame.

reiframe→get_errorType()

Returns the numerical error code of the latest error with the reference frame.

reiframe→get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME . FUNCTION_NAME`.

reiframe→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

reiframe→get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

reiframe→get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form `SERIAL . FUNCTIONID`.

reiframe→get_logicalName()

Returns the logical name of the reference frame.

reiframe→get_module()

Gets the `YModule` object for the device on which the function is located.

reiframe→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

reiframe→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.

reiframe→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.

reiframe→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

reiframe→isOnline()

Checks if the reference frame is currently reachable, without raising any error.

reiframe→isOnline_async(callback, context)

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reiframe→load(msValidity)

Preloads the reference frame cache with a specified validity duration.

reiframe→load_async(msValidity, callback, context)

Preloads the reference frame cache with a specified validity duration (asynchronous version).

reiframe→more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

reiframe→nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

reiframe→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

reiframe→save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

reiframe→set_bearing(newval)

Changes the reference bearing used by the compass.

reiframe→set_logicalName(newval)

Changes the logical name of the reference frame.

refframe→set_mountPosition(position, orientation)

Changes the compass and tilt sensor frame of reference.

refframe→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

refframe→start3DCalibration()

Initiates the sensors tridimensional calibration process.

refframe→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRefFrame.FindRefFrame() yFindRefFrame()yFindRefFrame()

YRefFrame

Retrieves a reference frame for a given identifier.

```
function yFindRefFrame( func: string): TYRefFrame
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the reference frame

Returns :

a `YRefFrame` object allowing you to drive the reference frame.

**YRefFrame.FirstRefFrame()
yFirstRefFrame()yFirstRefFrame()**

YRefFrame

Starts the enumeration of reference frames currently accessible.

```
function yFirstRefFrame( ): TYRefFrame
```

Use the method `YRefFrame.nextRefFrame()` to iterate on next reference frames.

Returns :

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

refframe→cancel3DCalibration()
refframe.cancel3DCalibration()

YRefFrame

Aborts the sensors tridimensional calibration process et restores normal settings.

function **cancel3DCalibration**(): LongInt

On failure, throws an exception or returns a negative error code.

refframe→describe()refframe.describe()**YRefFrame**

Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the reference frame (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

refframe→get_3DCalibrationHint()

YRefFrame

refframe→3DCalibrationHint()

refframe.get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

```
function get_3DCalibrationHint( ): string
```

Returns :

a character string.

refframe→get_3DCalibrationLogMsg()
refframe→3DCalibrationLogMsg()
refframe.get_3DCalibrationLogMsg()

YRefFrame

Returns the latest log message from the calibration process.

```
function get_3DCalibrationLogMsg( ): string
```

When no new message is available, returns an empty string.

Returns :

a character string.

refframe→get_3DCalibrationProgress()

YRefFrame

refframe→3DCalibrationProgress()

refframe.get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

function **get_3DCalibrationProgress**(): LongInt

Returns :

an integer between 0 (not started) and 100 (stage completed).

refframe→get_3DCalibrationStage()
refframe→3DCalibrationStage()
refframe.get_3DCalibrationStage()

YRefFrame

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

```
function get_3DCalibrationStage( ): LongInt
```

Returns :

an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress()

YRefFrame

refframe→3DCalibrationStageProgress()

refframe.get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

function **get_3DCalibrationStageProgress()**: LongInt

Returns :

an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue()**YRefFrame****refframe→advertisedValue()****refframe.get_advertisedValue()**

Returns the current value of the reference frame (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the reference frame (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

refframe→**get_bearing()**

YRefFrame

refframe→**bearing()****refframe.get_bearing()**

Returns the reference bearing used by the compass.

function **get_bearing**(): double

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

Returns :

a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns `Y_BEARING_INVALID`.

refframe→get_errorMessage()
refframe→errorMessage()
refframe.get_errorMessage()

YRefFrame

Returns the error message of the latest error with the reference frame.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the reference frame object

refframe→**get_errorType()**

YRefFrame

refframe→**errorType()****refframe.get_errorType()**

Returns the numerical error code of the latest error with the reference frame.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the reference frame object

refframe→get_functionDescriptor()
refframe→functionDescriptor()
refframe.get_functionDescriptor()

YRefFrame

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

refframe→**get_logicalName()**

YRefFrame

refframe→**logicalName()****refframe.get_logicalName()**

Returns the logical name of the reference frame.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the reference frame. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

refframe→**get_module()****YRefFrame****refframe**→**module()****refframe.get_module()**

Gets the `YModule` object for the device on which the function is located.

```
function get_module( ): YModule
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

Returns :

an instance of `YModule`

refframe→get_mountOrientation()**YRefFrame****refframe→mountOrientation()****refframe.get_mountOrientation()**

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

```
function get_mountOrientation( ): TYMOUNTORIENTATION
```

Returns :

a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

refframe→get_mountPosition()**YRefFrame****refframe→mountPosition()****refframe.get_mountPosition()**

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

```
function get_mountPosition( ): TYMOUNTPOSITION
```

Returns :

a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.

refframe→**get_userData()**

YRefFrame

refframe→**userData()****refframe.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

refframe→isOnline()refframe.isOnline()**YRefFrame**

Checks if the reference frame is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

Returns :

`true` if the reference frame can be reached, and `false` otherwise

refframe→load()**refframe.load()****YRefFrame**

Preloads the reference frame cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

refframe→more3DCalibration()
refframe.more3DCalibration()**YRefFrame**

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

```
function more3DCalibration( ): LongInt
```

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.

refframe→**nextRefFrame()****refframe.nextRefFrame()**

YRefFrame

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

function **nextRefFrame**(): TYRefFrame

Returns :

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a `null` pointer if there are no more reference frames to enumerate.

refframe→registerValueCallback()
refframe.registerValueCallback()

YRefFrame

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYRefFrameValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

refframe→save3DCalibration()
refframe.save3DCalibration()

YRefFrame

Applies the sensors tridimensional calibration parameters that have just been computed.

function **save3DCalibration**(): LongInt

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

refframe→**set_bearing()****YRefFrame****refframe**→**setBearing()****refframe.set_bearing()**

Changes the reference bearing used by the compass.

```
function set_bearing( newval: double): integer
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→**set_logicalName()**

YRefFrame

refframe→**setLogicalName()**

refframe.set_logicalName()

Changes the logical name of the reference frame.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the reference frame.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

refframe→set_mountPosition()
refframe→setMountPosition()
refframe.set_mountPosition()

YRefFrame

Changes the compass and tilt sensor frame of reference.

```
function set_mountPosition( position: TYMOUNTPOSITION,
                           orientation: TYMOUNTORIENTATION): LongInt
```

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

Parameters :

- position** a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.
- orientation** a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

refframe→set_userdata()

YRefFrame

refframe→setUserData()refframe.set_userdata()

Stores a user context provided as argument in the `userData` attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

refframe→start3DCalibration()
refframe.start3DCalibration()

YRefFrame

Initiates the sensors tridimensional calibration process.

```
function start3DCalibration( ): LongInt
```

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

3.35. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_relay.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YRelay = yoctolib.YRelay;</code>
php	<code>require_once('yocto_relay.php');</code>
c++	<code>#include "yocto_relay.h"</code>
m	<code>#import "yocto_relay.h"</code>
pas	<code>uses yocto_relay;</code>
vb	<code>yocto_relay.vb</code>
cs	<code>yocto_relay.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YRelay;</code>
py	<code>from yocto_relay import *</code>

Global functions

yFindRelay(func)

Retrieves a relay for a given identifier.

yFirstRelay()

Starts the enumeration of relays currently accessible.

YRelay methods

relay→delayedPulse(ms_delay, ms_duration)

Schedules a pulse.

relay→describe()

Returns a short text that describes unambiguously the instance of the relay in the form
TYPE (NAME) = SERIAL . FUNCTIONID.

relay→get_advertisedValue()

Returns the current value of the relay (no more than 6 characters).

relay→get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

relay→get_errorMessage()

Returns the error message of the latest error with the relay.

relay→get_errorType()

Returns the numerical error code of the latest error with the relay.

relay→get_friendlyName()

Returns a global identifier of the relay in the format MODULE_NAME . FUNCTION_NAME.

relay→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

relay→get_functionId()

Returns the hardware identifier of the relay, without reference to the module.

relay→get_hardwareId()

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

relay→get_logicalName()

Returns the logical name of the relay.

relay→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

relay→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

relay→get_module()

Gets the `YModule` object for the device on which the function is located.

relay→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

relay→get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

relay→get_pulseTimer()

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

relay→get_state()

Returns the state of the relays (A for the idle position, B for the active position).

relay→get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

relay→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

relay→isOnline()

Checks if the relay is currently reachable, without raising any error.

relay→isOnline_async(callback, context)

Checks if the relay is currently reachable, without raising any error (asynchronous version).

relay→load(msValidity)

Preloads the relay cache with a specified validity duration.

relay→load_async(msValidity, callback, context)

Preloads the relay cache with a specified validity duration (asynchronous version).

relay→nextRelay()

Continues the enumeration of relays started using `yFirstRelay()`.

relay→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

relay→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

relay→set_logicalName(newval)

Changes the logical name of the relay.

relay→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

relay→set_maxTimeOnStateB(newval)

3. Reference

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

relay→set_output(newval)

Changes the output state of the relays, when used as a simple switch (single throw).

relay→set_state(newval)

Changes the state of the relays (A for the idle position, B for the active position).

relay→set_stateAtPowerOn(newval)

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

relay→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

relay→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRelay.FindRelay() yFindRelay()yFindRelay()

YRelay

Retrieves a relay for a given identifier.

```
function yFindRelay( func: string): TYRelay
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the relay is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRelay.isOnline()` to test if the relay is indeed online at a given time. In case of ambiguity when looking for a relay by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the relay

Returns :

a `YRelay` object allowing you to drive the relay.

YRelay.FirstRelay()

YRelay

yFirstRelay()**yFirstRelay()**

Starts the enumeration of relays currently accessible.

```
function yFirstRelay( ): TYRelay
```

Use the method `YRelay.nextRelay()` to iterate on next relays.

Returns :

a pointer to a `YRelay` object, corresponding to the first relay currently online, or a `null` pointer if there are none.

relay→delayedPulse()**relay.delayedPulse()****YRelay**

Schedules a pulse.

```
function delayedPulse( ms_delay: LongInt, ms_duration: LongInt): integer
```

Parameters :

ms_delay waiting time before the pulse, in milliseconds

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→describe()relay.describe()**YRelay**

Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the relay (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

relay→get_advertisedValue()**YRelay****relay→advertisedValue()relay.get_advertisedValue()**

Returns the current value of the relay (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the relay (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

relay→**get_countdown()**

YRelay

relay→**countdown()****relay.get_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

```
function get_countdown( ): int64
```

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns Y_COUNTDOWN_INVALID.

relay→**get_errorMessage()****YRelay****relay**→**errorMessage()****relay.get_errorMessage()**

Returns the error message of the latest error with the relay.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the relay object

relay→**get_errorType()**

YRelay

relay→**errorType()****relay.get_errorType()**

Returns the numerical error code of the latest error with the relay.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the relay object

relay→**get_functionDescriptor()**
relay→**functionDescriptor()**
relay.get_functionDescriptor()

YRelay

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

relay→**get_logicalName()**

YRelay

relay→**logicalName()****relay.get_logicalName()**

Returns the logical name of the relay.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the relay. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

relay→**get_maxTimeOnStateA()****YRelay****relay**→**maxTimeOnStateA()****relay.get_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
function get_maxTimeOnStateA( ): int64
```

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.

relay→**get_maxTimeOnStateB()**

YRelay

relay→**maxTimeOnStateB()**

relay.get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
function get_maxTimeOnStateB( ): int64
```

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEB_INVALID.

relay→**get_module()****YRelay****relay**→**module()****relay.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

relay→**get_output()**

YRelay

relay→**output()****relay.get_output()**

Returns the output state of the relays, when used as a simple switch (single throw).

function **get_output**(): Integer

Returns :

either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.

relay→**get_pulseTimer()****YRelay****relay**→**pulseTimer()****relay.get_pulseTimer()**

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

```
function get_pulseTimer( ): int64
```

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

relay→**get_state()**

YRelay

relay→**state()****relay.get_state()**

Returns the state of the relays (A for the idle position, B for the active position).

function **get_state**(): Integer

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

relay→get_stateAtPowerOn()**YRelay****relay→stateAtPowerOn()relay.get_stateAtPowerOn()**

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```
function get_stateAtPowerOn( ): Integer
```

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

relay→**get_userData()**

YRelay

relay→**userData()****relay.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

relay→**isOnline()****relay.isOnline()****YRelay**

Checks if the relay is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the relay.

Returns :

`true` if the relay can be reached, and `false` otherwise

relay→load()relay.load()**YRelay**

Preloads the relay cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

relay→nextRelay()relay.nextRelay()

YRelay

Continues the enumeration of relays started using `yFirstRelay()`.

```
function nextRelay( ): TYRelay
```

Returns :

a pointer to a `YRelay` object, corresponding to a relay currently online, or a `null` pointer if there are no more relays to enumerate.

relay→pulse()relay.pulse()**YRelay**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
function pulse( ms_duration: LongInt): integer
```

Parameters :

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→registerValueCallback()
relay.registerValueCallback()

YRelay

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYRelayValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

relay→**set_logicalName()**

YRelay

relay→**setLogicalName()****relay.set_logicalName()**

Changes the logical name of the relay.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the relay.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateA()**YRelay****relay→setMaxTimeOnStateA()****relay.set_maxTimeOnStateA()**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
function set_maxTimeOnStateA( newval: int64): integer
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**set_maxTimeOnStateB()****YRelay****relay**→**setMaxTimeOnStateB()****relay.set_maxTimeOnStateB()**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
function set_maxTimeOnStateB( newval: int64): integer
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**set_output()****YRelay****relay**→**setOutput()****relay.set_output()**

Changes the output state of the relays, when used as a simple switch (single throw).

```
function set_output( newval: Integer): integer
```

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**set_state()**

YRelay

relay→**setState()****relay.set_state()**

Changes the state of the relays (A for the idle position, B for the active position).

```
function set_state( newval: Integer): integer
```

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_stateAtPowerOn()**YRelay****relay→setStateAtPowerOn()****relay.set_stateAtPowerOn()**

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
function set_stateAtPowerOn( newval: Integer): integer
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→**set_userData()**

YRelay

relay→**setUserData()****relay.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.36. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

Global functions

yFindSensor(func)

Retrieves a sensor for a given identifier.

yFirstSensor()

Starts the enumeration of sensors currently accessible.

YSensor methods

sensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

sensor→describe()

Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

sensor→get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

sensor→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

sensor→get_currentValue()

Returns the current value of the measure.

sensor→get_errorMessage()

Returns the error message of the latest error with the sensor.

sensor→get_errorType()

Returns the numerical error code of the latest error with the sensor.

sensor→get_friendlyName()

Returns a global identifier of the sensor in the format `MODULE_NAME . FUNCTION_NAME`.

sensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

sensor→get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

sensor→get_hardwareId()

3. Reference

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

sensor→**get_highestValue()**

Returns the maximal value observed for the measure since the device was started.

sensor→**get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

sensor→**get_logicalName()**

Returns the logical name of the sensor.

sensor→**get_lowestValue()**

Returns the minimal value observed for the measure since the device was started.

sensor→**get_module()**

Gets the `YModule` object for the device on which the function is located.

sensor→**get_module_async(callback, context)**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

sensor→**get_recordedData(startTime, endTime)**

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

sensor→**get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

sensor→**get_resolution()**

Returns the resolution of the measured values.

sensor→**get_unit()**

Returns the measuring unit for the measure.

sensor→**get_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

sensor→**isOnline()**

Checks if the sensor is currently reachable, without raising any error.

sensor→**isOnline_async(callback, context)**

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

sensor→**load(msValidity)**

Preloads the sensor cache with a specified validity duration.

sensor→**loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

sensor→**load_async(msValidity, callback, context)**

Preloads the sensor cache with a specified validity duration (asynchronous version).

sensor→**nextSensor()**

Continues the enumeration of sensors started using `yFirstSensor()`.

sensor→**registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

sensor→**registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

sensor→**set_highestValue(newval)**

Changes the recorded maximal value observed.

sensor→**set_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

sensor→**set_logicalName(newval)**

Changes the logical name of the sensor.

sensor→**set_lowestValue**(**newval**)

Changes the recorded minimal value observed.

sensor→**set_reportFrequency**(**newval**)

Changes the timed value notification frequency for this function.

sensor→**set_resolution**(**newval**)

Changes the resolution of the measured physical values.

sensor→**set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

sensor→**wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YSensor.FindSensor() yFindSensor()yFindSensor()

YSensor

Retrieves a sensor for a given identifier.

```
function yFindSensor( func: string): TYSensor
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the sensor

Returns :

a `YSensor` object allowing you to drive the sensor.

**YSensor.FirstSensor()
yFirstSensor()yFirstSensor()**

YSensor

Starts the enumeration of sensors currently accessible.

```
function yFirstSensor( ): TYSensor
```

Use the method `YSensor.nextSensor()` to iterate on next sensors.

Returns :

a pointer to a `YSensor` object, corresponding to the first sensor currently online, or a `null` pointer if there are none.

sensor→calibrateFromPoints()
sensor.calibrateFromPoints()**YSensor**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                               refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→describe()sensor.describe()**YSensor**

Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

sensor→**get_advertisedValue()**

YSensor

sensor→**advertisedValue()**

sensor.get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

sensor→**get_currentRawValue()****YSensor****sensor**→**currentRawValue()****sensor.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

sensor→**get_currentValue()**

YSensor

sensor→**currentValue()****sensor.get_currentValue()**

Returns the current value of the measure.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the measure

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

sensor→**get_errorMessage()****YSensor****sensor**→**errorMessage()****sensor.get_errorMessage()**

Returns the error message of the latest error with the sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the sensor object

sensor→**get_errorType()**

YSensor

sensor→**errorType()****sensor.get_errorType()**

Returns the numerical error code of the latest error with the sensor.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the sensor object

sensor→**get_functionDescriptor()**
sensor→**functionDescriptor()**
sensor.get_functionDescriptor()

YSensor

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

sensor→**get_highestValue()**

YSensor

sensor→**highestValue()****sensor.get_highestValue()**

Returns the maximal value observed for the measure since the device was started.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

sensor→get_logFrequency()**YSensor****sensor→logFrequency()sensor.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

sensor→**get_logicalName()**

YSensor

sensor→**logicalName()****sensor.get_logicalName()**

Returns the logical name of the sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

sensor→**get_lowestValue()****YSensor****sensor**→**lowestValue()****sensor.get_lowestValue()**

Returns the minimal value observed for the measure since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

sensor→**get_module()**

YSensor

sensor→**module()****sensor.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module()**: TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

sensor→**get_recordedData()****YSensor****sensor**→**recordedData()****sensor.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

sensor→get_reportFrequency()

YSensor

sensor→reportFrequency()

sensor.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

sensor→**get_resolution()****YSensor****sensor**→**resolution()****sensor.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

sensor→**get_unit()**

YSensor

sensor→**unit()****sensor.get_unit()**

Returns the measuring unit for the measure.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

sensor→**get_userData()****YSensor****sensor**→**userData()****sensor.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

sensor→**isOnline()****sensor.isOnline()**

YSensor

Checks if the sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the sensor.

Returns :

`true` if the sensor can be reached, and `false` otherwise

sensor→load()**sensor.load()****YSensor**

Preloads the sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

sensor→loadCalibrationPoints()
sensor.loadCalibrationPoints()**YSensor**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**nextSensor()****sensor.nextSensor()****YSensor**

Continues the enumeration of sensors started using `yFirstSensor()`.

```
function nextSensor( ): TYSensor
```

Returns :

a pointer to a `YSensor` object, corresponding to a sensor currently online, or a `null` pointer if there are no more sensors to enumerate.

sensor→registerTimedReportCallback()
sensor.registerTimedReportCallback()**YSensor**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYSensorTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

sensor→registerValueCallback()
sensor.registerValueCallback()

YSensor

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYSensorValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

sensor→**set_highestValue()**

YSensor

sensor→**setHighestValue()****sensor.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_logFrequency()****YSensor****sensor**→**setLogFrequency()****sensor.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_logicalName()**

YSensor

sensor→**setLogicalName()****sensor.set_logicalName()**

Changes the logical name of the sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

sensor→**set_lowestValue()****YSensor****sensor**→**setLowestValue()****sensor.set_lowestValue()**

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_reportFrequency()****YSensor****sensor**→**setReportFrequency()****sensor.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_resolution()****YSensor****sensor**→**setResolution()****sensor.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→**set_userData()**

YSensor

sensor→**setUserData()****sensor.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.37. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_servo.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YServo = yoctolib.YServo;
php	require_once('yocto_servo.php');
c++	#include "yocto_servo.h"
m	#import "yocto_servo.h"
pas	uses yocto_servo;
vb	yocto_servo.vb
cs	yocto_servo.cs
java	import com.yoctopuce.YoctoAPI.YServo;
py	from yocto_servo import *

Global functions

yFindServo(func)

Retrieves a servo for a given identifier.

yFirstServo()

Starts the enumeration of servos currently accessible.

YServo methods

servo→describe()

Returns a short text that describes unambiguously the instance of the servo in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

servo→get_advertisedValue()

Returns the current value of the servo (no more than 6 characters).

servo→get_enabled()

Returns the state of the servos.

servo→get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

servo→get_errorMessage()

Returns the error message of the latest error with the servo.

servo→get_errorType()

Returns the numerical error code of the latest error with the servo.

servo→get_friendlyName()

Returns a global identifier of the servo in the format `MODULE_NAME . FUNCTION_NAME`.

servo→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

servo→get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

servo→get_hardwareId()

Returns the unique hardware identifier of the servo in the form `SERIAL . FUNCTIONID`.

servo→get_logicalName()

Returns the logical name of the servo.

servo→get_module()

Gets the YModule object for the device on which the function is located.

servo→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_positionAtPowerOn()

Returns the servo position at device power up.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

servo→isOnline()

Checks if the servo is currently reachable, without raising any error.

servo→isOnline_async(callback, context)

Checks if the servo is currently reachable, without raising any error (asynchronous version).

servo→load(msValidity)

Preloads the servo cache with a specified validity duration.

servo→load_async(msValidity, callback, context)

Preloads the servo cache with a specified validity duration (asynchronous version).

servo→move(target, ms_duration)

Performs a smooth move at constant speed toward a given position.

servo→nextServo()

Continues the enumeration of servos started using yFirstServo().

servo→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

servo→set_enabled(newval)

Stops or starts the servo.

servo→set_enabledAtPowerOn(newval)

Configure the servo signal generator state at power up.

servo→set_logicalName(newval)

Changes the logical name of the servo.

servo→set_neutral(newval)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→set_position(newval)

Changes immediately the servo driving position.

servo→set_positionAtPowerOn(newval)

Configure the servo position at device power up.

servo→set_range(newval)

Changes the range of use of the servo, specified in per cents.

servo→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

servo→wait_async(callback, context)

<p>Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.</p>

YServo.FindServo()**YServo****yFindServo()**

Retrieves a servo for a given identifier.

```
function yFindServo( func: string): TYServo
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the servo is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YServo.isOnline()` to test if the servo is indeed online at a given time. In case of ambiguity when looking for a servo by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the servo

Returns :

a `YServo` object allowing you to drive the servo.

**YServo.FirstServo()
yFirstServo()yFirstServo()**

YServo

Starts the enumeration of servos currently accessible.

```
function yFirstServo( ): TYServo
```

Use the method `YServo.nextServo()` to iterate on next servos.

Returns :

a pointer to a `YServo` object, corresponding to the first servo currently online, or a `null` pointer if there are none.

servo→describe()**servo.describe()****YServo**

Returns a short text that describes unambiguously the instance of the servo in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the servo (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

servo→**get_advertisedValue()****YServo****servo**→**advertisedValue()****servo.get_advertisedValue()**

Returns the current value of the servo (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the servo (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

servo→**get_enabled()**

YServo

servo→**enabled()****servo.get_enabled()**

Returns the state of the servos.

function **get_enabled**(): Integer

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the servos

On failure, throws an exception or returns Y_ENABLED_INVALID.

servo→get_enabledAtPowerOn()**YServo****servo→enabledAtPowerOn()****servo.get_enabledAtPowerOn()**

Returns the servo signal generator state at power up.

```
function get_enabledAtPowerOn( ): Integer
```

Returns :

either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the servo signal generator state at power up

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.

servo→**get_errorMessage()**

YServo

servo→**errorMessage()****servo.get_errorMessage()**

Returns the error message of the latest error with the servo.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the servo object

servo→get_errorType()**YServo****servo→errorType()****servo.get_errorType()**

Returns the numerical error code of the latest error with the servo.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the servo object

servo→get_functionDescriptor()

YServo

servo→functionDescriptor()

servo.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

servo→get_logicalName()**YServo****servo→logicalName()****servo.get_logicalName()**

Returns the logical name of the servo.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the servo. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

servo→**get_module()**

YServo

servo→**module()****servo.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module**(): TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

servo→get_neutral()**YServo****servo→neutral()servo.get_neutral()**

Returns the duration in microseconds of a neutral pulse for the servo.

function **get_neutral**(): LongInt

Returns :

an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns Y_NEUTRAL_INVALID.

servo→get_position()

YServo

servo→position()**servo.get_position()**

Returns the current servo position.

function **get_position**(): LongInt

Returns :

an integer corresponding to the current servo position

On failure, throws an exception or returns Y_POSITION_INVALID.

servo→get_positionAtPowerOn()**YServo****servo→positionAtPowerOn()****servo.get_positionAtPowerOn()**

Returns the servo position at device power up.

```
function get_positionAtPowerOn( ): LongInt
```

Returns :

an integer corresponding to the servo position at device power up

On failure, throws an exception or returns Y_POSITIONATPOWERON_INVALID.

servo→get_range()

YServo

servo→range()servo.get_range()

Returns the current range of use of the servo.

function **get_range**(): LongInt

Returns :

an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns Y_RANGE_INVALID.

servo→get_userdata()**YServo****servo→userData()servo.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

function **get_userdata()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

servo→isOnline()**servo.isOnline()**

YServo

Checks if the servo is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

Returns :

`true` if the servo can be reached, and `false` otherwise

servo→load()**servo.load()****YServo**

Preloads the servo cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

servo→move()**servo.move()****YServo**

Performs a smooth move at constant speed toward a given position.

```
function move( target: LongInt, ms_duration: LongInt): integer
```

Parameters :

target new position at the end of the move
ms_duration total duration of the move, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→nextServo()**servo.nextServo()****YServo**

Continues the enumeration of servos started using `yFirstServo()`.

```
function nextServo( ): TYServo
```

Returns :

a pointer to a `YServo` object, corresponding to a servo currently online, or a `null` pointer if there are no more servos to enumerate.

servo→registerValueCallback()
servo.registerValueCallback()**YServo**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYServoValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

servo→**set_enabled()****YServo****servo**→**setEnabled()****servo.set_enabled()**

Stops or starts the servo.

```
function set_enabled( newval: Integer): integer
```

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_enabledAtPowerOn()

YServo

servo→setEnabledAtPowerOn()

servo.set_enabledAtPowerOn()

Configure the servo signal generator state at power up.

```
function set_enabledAtPowerOn( newval: Integer): integer
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→**set_logicalName()****YServo****servo**→**setLogicalName()****servo.set_logicalName()**

Changes the logical name of the servo.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the servo.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

servo→**set_neutral()****YServo****servo**→**setNeutral()****servo.set_neutral()**

Changes the duration of the pulse corresponding to the neutral position of the servo.

```
function set_neutral( newval: LongInt): integer
```

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→**set_position()****YServo****servo**→**setPosition()****servo.set_position()**

Changes immediately the servo driving position.

```
function set_position( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to immediately the servo driving position

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_positionAtPowerOn()
servo→setPositionAtPowerOn()
servo.set_positionAtPowerOn()

YServo

Configure the servo position at device power up.

```
function set_positionAtPowerOn( newval: LongInt): integer
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→**set_range()****YServo****servo**→**setRange()****servo.set_range()**

Changes the range of use of the servo, specified in per cents.

```
function set_range( newval: LongInt): integer
```

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms], When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the range of use of the servo, specified in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→**set_userData()**

YServo

servo→**setUserData()****servo.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.38. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_temperature.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YTemperature = yoctolib.YTemperature;
php	require_once('yocto_temperature.php');
c++	#include "yocto_temperature.h"
m	#import "yocto_temperature.h"
pas	uses yocto_temperature;
vb	yocto_temperature.vb
cs	yocto_temperature.cs
java	import com.yoctopuce.YoctoAPI.YTemperature;
py	from yocto_temperature import *

Global functions

yFindTemperature(func)

Retrieves a temperature sensor for a given identifier.

yFirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

YTemperature methods

temperature→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

temperature→describe()

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

temperature→get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

temperature→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

temperature→get_currentValue()

Returns the current value of the temperature.

temperature→get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

temperature→get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

temperature→get_friendlyName()

Returns a global identifier of the temperature sensor in the format `MODULE_NAME . FUNCTION_NAME`.

temperature→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

temperature→get_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

temperature→get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL . FUNCTIONID`.

temperature→get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

temperature→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

temperature→get_logicalName()

Returns the logical name of the temperature sensor.

temperature→get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

temperature→get_module()

Gets the YModule object for the device on which the function is located.

temperature→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

temperature→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

temperature→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

temperature→get_resolution()

Returns the resolution of the measured values.

temperature→get_sensorType()

Returns the temperature sensor type.

temperature→get_unit()

Returns the measuring unit for the temperature.

temperature→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

temperature→isOnline()

Checks if the temperature sensor is currently reachable, without raising any error.

temperature→isOnline_async(callback, context)

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

temperature→load(msValidity)

Preloads the temperature sensor cache with a specified validity duration.

temperature→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

temperature→load_async(msValidity, callback, context)

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

temperature→nextTemperature()

Continues the enumeration of temperature sensors started using yFirstTemperature().

temperature→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

temperature→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

temperature→set_highestValue(newval)

Changes the recorded maximal value observed.

temperature→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

temperature→set_logicalName(newval)

Changes the logical name of the temperature sensor.

temperature→set_lowestValue(newval)

Changes the recorded minimal value observed.

temperature→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

temperature→set_resolution(newval)

Changes the resolution of the measured physical values.

temperature→set_sensorType(newval)

Modify the temperature sensor type.

temperature→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

temperature→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTemperature.FindTemperature() yFindTemperature()yFindTemperature()

YTemperature

Retrieves a temperature sensor for a given identifier.

```
function yFindTemperature( func: string): TYTemperature
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the temperature sensor

Returns :

a YTemperature object allowing you to drive the temperature sensor.

**YTemperature.FirstTemperature()
yFirstTemperature()yFirstTemperature()**

YTemperature

Starts the enumeration of temperature sensors currently accessible.

```
function yFirstTemperature( ): TYTemperature
```

Use the method `YTemperature.nextTemperature()` to iterate on next temperature sensors.

Returns :

a pointer to a `YTemperature` object, corresponding to the first temperature sensor currently online, or a `null` pointer if there are none.

**temperature→calibrateFromPoints()
temperature.calibrateFromPoints()****YTemperature**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                               refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→describe()temperature.describe()**YTemperature**

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the temperature sensor (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

temperature→**get_advertisedValue()**

YTemperature

temperature→**advertisedValue()**

temperature.get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the temperature sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

temperature→**get_currentRawValue()****YTemperature****temperature**→**currentRawValue()****temperature.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

temperature→get_currentValue()

YTemperature

temperature→currentValue()

temperature.get_currentValue()

Returns the current value of the temperature.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the temperature

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

temperature→**get_errorMessage()****YTemperature****temperature**→**errorMessage()****temperature.get_errorMessage()**

Returns the error message of the latest error with the temperature sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the temperature sensor object

temperature→get_errorType()
temperature→errorType()
temperature.get_errorType()

YTemperature

Returns the numerical error code of the latest error with the temperature sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the temperature sensor object

temperature→**get_functionDescriptor()**
temperature→**functionDescriptor()**
temperature.get_functionDescriptor()

YTemperature

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

temperature→get_highestValue()

YTemperature

temperature→highestValue()

temperature.get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

temperature→get_logFrequency()**YTemperature****temperature→logFrequency()****temperature.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

temperature→get_logicalName()

YTemperature

temperature→logicalName()

temperature.get_logicalName()

Returns the logical name of the temperature sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the temperature sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

temperature→**get_lowestValue()****YTemperature****temperature**→**lowestValue()****temperature.get_lowestValue()**

Returns the minimal value observed for the temperature since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

temperature→get_module()

YTemperature

temperature→module()temperature.get_module()

Gets the YModule object for the device on which the function is located.

function **get_module()**: TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

temperature→get_recordedData()**YTemperature****temperature→recordedData()****temperature.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

temperature→get_reportFrequency()

YTemperature

temperature→reportFrequency()

temperature.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

temperature→get_resolution()**YTemperature****temperature→resolution()****temperature.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

temperature→get_sensorType()

YTemperature

temperature→sensorType()

temperature.get_sensorType()

Returns the temperature sensor type.

function **get_sensorType**(): Integer

Returns :

a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES corresponding to the temperature sensor type

On failure, throws an exception or returns Y_SENSORTYPE_INVALID.

temperature→**get_unit()****YTemperature****temperature**→**unit()****temperature.get_unit()**

Returns the measuring unit for the temperature.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the temperature

On failure, throws an exception or returns Y_UNIT_INVALID.

temperature→**get_userData()**

YTemperature

temperature→**userData()****temperature.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

temperature→**isOnline()****temperature.isOnline()****YTemperature**

Checks if the temperature sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

Returns :

`true` if the temperature sensor can be reached, and `false` otherwise

temperature→load()temperature.load()**YTemperature**

Preloads the temperature sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**temperature→loadCalibrationPoints()
temperature.loadCalibrationPoints()**

YTemperature

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→nextTemperature()
temperature.nextTemperature()

YTemperature

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

function **nextTemperature()**: TYTemperature

Returns :

a pointer to a YTemperature object, corresponding to a temperature sensor currently online, or a null pointer if there are no more temperature sensors to enumerate.

temperature→**registerTimedReportCallback()**
temperature.registerTimedReportCallback()

YTemperature

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYTemperatureTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

temperature→registerValueCallback()
temperature.registerValueCallback()

YTemperature

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYTemperatureValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

temperature→**set_highestValue()****YTemperature****temperature**→**setHighestValue()****temperature.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logFrequency()

YTemperature

temperature→setLogFrequency()

temperature.set_logFrequency()

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_logicalName()****YTemperature****temperature**→**setLogicalName()****temperature.set_logicalName()**

Changes the logical name of the temperature sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the temperature sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

temperature→set_lowestValue()

YTemperature

temperature→setLowestValue()

temperature.set_lowestValue()

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_reportFrequency()**
temperature→**setReportFrequency()**
temperature.set_reportFrequency()

YTemperature

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_resolution()

YTemperature

temperature→setResolution()

temperature.set_resolution()

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→**set_sensorType()****YTemperature****temperature**→**setSensorType()****temperature.set_sensorType()**

Modify the temperature sensor type.

```
function set_sensorType( newval: Integer): integer
```

This function is used to to define the type of thermocouple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_userdata()

YTemperature

temperature→setUserData()

temperature.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.39. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_tilt.js'></script></code>
nodejs	<code>var yoctolib = require('yoctolib'); var YTilt = yoctolib.YTilt;</code>
php	<code>require_once('yocto_tilt.php');</code>
c++	<code>#include "yocto_tilt.h"</code>
m	<code>#import "yocto_tilt.h"</code>
pas	<code>uses yocto_tilt;</code>
vb	<code>yocto_tilt.vb</code>
cs	<code>yocto_tilt.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YTilt;</code>
py	<code>from yocto_tilt import *</code>

Global functions

yFindTilt(func)

Retrieves a tilt sensor for a given identifier.

yFirstTilt()

Starts the enumeration of tilt sensors currently accessible.

YTilt methods

tilt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

tilt→describe()

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

tilt→get_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

tilt→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

tilt→get_currentValue()

Returns the current value of the inclination.

tilt→get_errorMessage()

Returns the error message of the latest error with the tilt sensor.

tilt→get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

tilt→get_friendlyName()

Returns a global identifier of the tilt sensor in the format `MODULE_NAME . FUNCTION_NAME`.

tilt→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

tilt→get_functionId()

Returns the hardware identifier of the tilt sensor, without reference to the module.

tilt→get_hardwareId()

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL . FUNCTIONID`.

tilt→get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

tilt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

tilt→get_logicalName()

Returns the logical name of the tilt sensor.

tilt→get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

tilt→get_module()

Gets the YModule object for the device on which the function is located.

tilt→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

tilt→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

tilt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

tilt→get_resolution()

Returns the resolution of the measured values.

tilt→get_unit()

Returns the measuring unit for the inclination.

tilt→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

tilt→isOnline()

Checks if the tilt sensor is currently reachable, without raising any error.

tilt→isOnline_async(callback, context)

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

tilt→load(msValidity)

Preloads the tilt sensor cache with a specified validity duration.

tilt→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

tilt→load_async(msValidity, callback, context)

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

tilt→nextTilt()

Continues the enumeration of tilt sensors started using yFirstTilt().

tilt→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

tilt→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

tilt→set_highestValue(newval)

Changes the recorded maximal value observed.

tilt→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

tilt→set_logicalName(newval)

Changes the logical name of the tilt sensor.

tilt→set_lowestValue(newval)

Changes the recorded minimal value observed.

tilt→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

tilt→set_resolution(newval)

Changes the resolution of the measured physical values.

tilt→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

tilt→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YTilt.FindTilt() yFindTilt()

YTilt

Retrieves a tilt sensor for a given identifier.

```
function yFindTilt( func: string): YTilt
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the tilt sensor

Returns :

a `YTilt` object allowing you to drive the tilt sensor.

YTilt.FirstTilt()**YTilt****yFirstTilt()****yFirstTilt()**

Starts the enumeration of tilt sensors currently accessible.

```
function yFirstTilt( ): TYTilt
```

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

Returns :

a pointer to a `YTilt` object, corresponding to the first tilt sensor currently online, or a `null` pointer if there are none.

tilt→calibrateFromPoints()**tilt.calibrateFromPoints()****YTilt**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→describe()tilt.describe()**YTilt**

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the tilt sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

tilt→**get_advertisedValue()**

YTilt

tilt→**advertisedValue()****tilt.get_advertisedValue()**

Returns the current value of the tilt sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the tilt sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

tilt→**get_currentRawValue()****YTilt****tilt**→**currentRawValue()****tilt.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

tilt→**get_currentValue()**

YTilt

tilt→**currentValue()****tilt.get_currentValue()**

Returns the current value of the inclination.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current value of the inclination

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

tilt→**get_errorMessage()****YTilt****tilt**→**errorMessage()****tilt.get_errorMessage()**

Returns the error message of the latest error with the tilt sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the tilt sensor object

tilt→get_errorType()

YTilt

tilt→errorType()tilt.get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the tilt sensor object

tilt→**get_functionDescriptor()****YTilt****tilt**→**functionDescriptor()****tilt.get_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

tilt→**get_highestValue()**

YTilt

tilt→**highestValue()****tilt.get_highestValue()**

Returns the maximal value observed for the inclination since the device was started.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

tilt→get_logFrequency()**YTilt****tilt→logFrequency()tilt.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

tilt→**get_logicalName()**

YTilt

tilt→**logicalName()****tilt.get_logicalName()**

Returns the logical name of the tilt sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the tilt sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

tilt→**get_lowestValue()****YTilt****tilt**→**lowestValue()****tilt.get_lowestValue()**

Returns the minimal value observed for the inclination since the device was started.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

tilt→get_module()

YTilt

tilt→module()tilt.get_module()

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TYModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

tilt→**get_recordedData()****YTilt****tilt**→**recordedData()****tilt.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

tilt→get_reportFrequency()

YTilt

tilt→reportFrequency()**tilt.get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

tilt→**get_resolution()****YTilt****tilt**→**resolution()****tilt.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

tilt→**get_unit()**

YTilt

tilt→**unit()****tilt.get_unit()**

Returns the measuring unit for the inclination.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns `Y_UNIT_INVALID`.

tilt→**get_userData()****YTilt****tilt**→**userData()****tilt.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

tilt→isOnline()tilt.isOnline()**YTilt**

Checks if the tilt sensor is currently reachable, without raising any error.

```
function isOnline(): boolean
```

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

Returns :

`true` if the tilt sensor can be reached, and `false` otherwise

tilt→load()tilt.load()**YTilt**

Preloads the tilt sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

tilt→loadCalibrationPoints()
tilt.loadCalibrationPoints()**YTilt**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

- rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.
- refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→**nextTilt()****tilt.nextTilt()****YTilt**

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

```
function nextTilt( ): YTilt
```

Returns :

a pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a `null` pointer if there are no more tilt sensors to enumerate.

tilt→registerTimedReportCallback()
tilt.registerTimedReportCallback()**YTilt**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: YTiltTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

tilt→registerValueCallback()
tilt.registerValueCallback()

YTilt

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYTiltValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

tilt→**set_highestValue()****YTilt****tilt**→**setHighestValue()****tilt.set_highestValue()**

Changes the recorded maximal value observed.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→**set_logFrequency()****YTilt****tilt**→**setLogFrequency()****tilt.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→**set_logicalName()****YTilt****tilt**→**setLogicalName()****tilt.set_logicalName()**

Changes the logical name of the tilt sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the tilt sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

tilt→**set_lowestValue()****YTilt****tilt**→**setLowestValue()****tilt.set_lowestValue()**

Changes the recorded minimal value observed.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_reportFrequency()**YTilt****tilt→setReportFrequency()tilt.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→**set_resolution()****YTilt****tilt**→**setResolution()****tilt.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_userdata()

YTilt

tilt→setUserData()tilt.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.40. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voc.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YVoc = yoctolib.YVoc;
php	require_once('yocto_voc.php');
c++	#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 unrounded and uncalibrated raw value returned by the sensor.

voc→get_currentValue()

Returns the current measure for the estimated VOC concentration.

voc→get_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

voc→get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

voc→get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format MODULE_NAME . FUNCTION_NAME.

voc→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

voc→get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

voc→get_hardwareId()

3. Reference

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

`voc→get_highestValue()`

Returns the maximal value observed for the estimated VOC concentration.

`voc→get_logFrequency()`

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

`voc→get_logicalName()`

Returns the logical name of the Volatile Organic Compound sensor.

`voc→get_lowestValue()`

Returns the minimal value observed for the estimated VOC concentration.

`voc→get_module()`

Gets the `YModule` object for the device on which the function is located.

`voc→get_module_async(callback, context)`

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`voc→get_recordedData(startTime, endTime)`

Retrieves a `DataSet` object holding historical data for this sensor, for a specified time interval.

`voc→get_reportFrequency()`

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

`voc→get_resolution()`

Returns the resolution of the measured values.

`voc→get_unit()`

Returns the measuring unit for the estimated VOC concentration.

`voc→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

`voc→isOnline()`

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

`voc→isOnline_async(callback, context)`

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

`voc→load(msValidity)`

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

`voc→loadCalibrationPoints(rawValues, refValues)`

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

`voc→load_async(msValidity, callback, context)`

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

`voc→nextVoc()`

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

`voc→registerTimedReportCallback(callback)`

Registers the callback function that is invoked on every periodic timed notification.

`voc→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`voc→set_highestValue(newval)`

Changes the recorded maximal value observed for the estimated VOC concentration.

voc→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voc→set_logicalName(newval)

Changes the logical name of the Volatile Organic Compound sensor.

voc→set_lowestValue(newval)

Changes the recorded minimal value observed for the estimated VOC concentration.

voc→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voc→set_resolution(newval)

Changes the resolution of the measured physical values.

voc→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voc→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoc.FindVoc() yFindVoc()yFindVoc()

YVoc

Retrieves a Volatile Organic Compound sensor for a given identifier.

```
function yFindVoc( func: string): TYVoc
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Volatile Organic Compound sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoc.isOnline()` to test if the Volatile Organic Compound sensor is indeed online at a given time. In case of ambiguity when looking for a Volatile Organic Compound sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the Volatile Organic Compound sensor

Returns :

a `YVoc` object allowing you to drive the Volatile Organic Compound sensor.

YVoc.FirstVoc() yFirstVoc()yFirstVoc()

YVoc

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

```
function yFirstVoc( ): TYVoc
```

Use the method `YVoc.nextVoc()` to iterate on next Volatile Organic Compound sensors.

Returns :

a pointer to a `YVoc` object, corresponding to the first Volatile Organic Compound sensor currently online, or a `null` pointer if there are none.

voc→calibrateFromPoints()**voc.calibrateFromPoints()****YVoc**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→describe()**voc.describe()****YVoc**

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the Volatile Organic Compound sensor (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voc→**get_advertisedValue()**

YVoc

voc→**advertisedValue()****vac.get_advertisedValue()**

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

voc→**get_currentRawValue()****YVoc****voc**→**currentRawValue()****voc.get_currentRawValue()**

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

voc→**get_currentValue()**

YVoc

voc→**currentValue()****voc.get_currentValue()**

Returns the current measure for the estimated VOC concentration.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current measure for the estimated VOC concentration

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

voc→**get_errorMessage()****YVoc****voc**→**errorMessage()****voc.get_errorMessage()**

Returns the error message of the latest error with the Volatile Organic Compound sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the Volatile Organic Compound sensor object

voc→**get_errorType()**

YVoc

voc→**errorType()****voc.get_errorType()**

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the Volatile Organic Compound sensor object

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voc→**get_highestValue()**

YVoc

voc→**highestValue()****voc.get_highestValue()**

Returns the maximal value observed for the estimated VOC concentration.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the estimated VOC concentration

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

voc→get_logFrequency()**YVoc****voc→logFrequency()voc.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

voc→**get_logicalName()**

YVoc

voc→**logicalName()****voc.get_logicalName()**

Returns the logical name of the Volatile Organic Compound sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the Volatile Organic Compound sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

voc→**get_lowestValue()****YVoc****voc**→**lowestValue()****voc.get_lowestValue()**

Returns the minimal value observed for the estimated VOC concentration.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the estimated VOC concentration

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

voc→**get_module()**

YVoc

voc→**module()****voc.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module**(): TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

voc→**get_recordedData()****YVoc****voc**→**recordedData()****voc.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voc→**get_reportFrequency()**

YVoc

voc→**reportFrequency()****voc.get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

voc→**get_resolution()****YVoc****voc**→**resolution()****voc.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

voc→**get_unit()**

YVoc

voc→**unit()****voc.get_unit()**

Returns the measuring unit for the estimated VOC concentration.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voc→**get_userData()****YVoc****voc**→**userData()****voc.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

voc→**isOnline()****voc.isOnline()****YVoc**

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Volatile Organic Compound sensor.

Returns :

`true` if the Volatile Organic Compound sensor can be reached, and `false` otherwise

voc→load()**voc.load()****YVoc**

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

voc→loadCalibrationPoints()
voc.loadCalibrationPoints()**YVoc**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

`voc→nextVoc()`**`voc.nextVoc()`****YVoc**

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

```
function nextVoc( ): TYVoc
```

Returns :

a pointer to a `YVoc` object, corresponding to a Volatile Organic Compound sensor currently online, or a `null` pointer if there are no more Volatile Organic Compound sensors to enumerate.

voc→**registerTimedReportCallback()****YVoc****voc.registerTimedReportCallback()**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYVocTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an YMeasure object describing the new advertised value.

voc→registerValueCallback()
voc.registerValueCallback()

YVoc

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYVocValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voc→**set_highestValue()**

YVoc

voc→**setHighestValue()****voc.set_highestValue()**

Changes the recorded maximal value observed for the estimated VOC concentration.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the estimated VOC concentration

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→**set_logFrequency()****YVoc****voc**→**setLogFrequency()****vac.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

YVoc
voc→**set_logicalName()****voc**→**setLogicalName()****voc.set_logicalName()**

Changes the logical name of the Volatile Organic Compound sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the Volatile Organic Compound sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

voc→**set_lowestValue()****YVoc****voc**→**setLowestValue()****voc.set_lowestValue()**

Changes the recorded minimal value observed for the estimated VOC concentration.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the estimated VOC concentration

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vac→set_reportFrequency()**YVoc****vac→setReportFrequency()****vac.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→**set_resolution()****YVoc****voc**→**setResolution()****voc.set_resolution()**

Changes the resolution of the measured physical values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→**set_userData()**

YVoc

voc→**setUserData()****voc.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.41. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voltage.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YVoltage = yoctolib.YVoltage;
php	require_once('yocto_voltage.php');
c++	#include "yocto_voltage.h"
m	#import "yocto_voltage.h"
pas	uses yocto_voltage;
vb	yocto_voltage.vb
cs	yocto_voltage.cs
java	import com.yoctopuce.YoctoAPI.YVoltage;
py	from yocto_voltage import *

Global functions

yFindVoltage(func)

Retrieves a voltage sensor for a given identifier.

yFirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

YVoltage methods

voltage→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voltage→describe()

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

voltage→get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

voltage→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

voltage→get_currentValue()

Returns the current measure for the voltage.

voltage→get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

voltage→get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

voltage→get_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME . FUNCTION_NAME`.

voltage→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voltage→get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

voltage→get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL . FUNCTIONID`.

voltage→get_highestValue()

Returns the maximal value observed for the voltage.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage.

voltage→get_module()

Gets the YModule object for the device on which the function is located.

voltage→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(callback, context)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→load(msValidity)

Preloads the voltage sensor cache with a specified validity duration.

voltage→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

voltage→load_async(msValidity, callback, context)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using yFirstVoltage().

voltage→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_highestValue(newval)

Changes the recorded maximal value observed pour the voltage.

voltage→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(newval)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed pour the voltage.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured values.

voltage→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltage.FindVoltage() yFindVoltage()yFindVoltage()

YVoltage

Retrieves a voltage sensor for a given identifier.

```
function yFindVoltage( func: string): TYVoltage
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.IsOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

**YVoltage.FirstVoltage()
yFirstVoltage()yFirstVoltage()**

YVoltage

Starts the enumeration of voltage sensors currently accessible.

```
function yFirstVoltage( ): TYVoltage
```

Use the method `YVoltage.nextVoltage()` to iterate on next voltage sensors.

Returns :

a pointer to a `YVoltage` object, corresponding to the first voltage sensor currently online, or a `null` pointer if there are none.

voltage→calibrateFromPoints()
voltage.calibrateFromPoints()**YVoltage**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
function calibrateFromPoints( rawValues: TDoubleArray,  
                             refValues: TDoubleArray): LongInt
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→describe()**voltage.describe()****YVoltage**

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voltage→**get_advertisedValue()**

YVoltage

voltage→**advertisedValue()**

voltage.get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the voltage sensor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

voltage→**get_currentRawValue()****YVoltage****voltage**→**currentRawValue()****voltage.get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( ): double
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

voltage→**get_currentValue()**

YVoltage

voltage→**currentValue()****voltage.get_currentValue()**

Returns the current measure for the voltage.

```
function get_currentValue( ): double
```

Returns :

a floating point number corresponding to the current measure for the voltage

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

voltage→**get_errorMessage()****YVoltage****voltage**→**errorMessage()****voltage.get_errorMessage()**

Returns the error message of the latest error with the voltage sensor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the voltage sensor object

voltage→**get_errorType()**

YVoltage

voltage→**errorType()****voltage.get_errorType()**

Returns the numerical error code of the latest error with the voltage sensor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the voltage sensor object

voltage→**get_functionDescriptor()****YVoltage****voltage**→**functionDescriptor()****voltage.get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

voltage→**get_highestValue()**

YVoltage

voltage→**highestValue()****voltage.get_highestValue()**

Returns the maximal value observed for the voltage.

```
function get_highestValue( ): double
```

Returns :

a floating point number corresponding to the maximal value observed for the voltage

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

voltage→**get_logFrequency()****YVoltage****voltage**→**logFrequency()****voltage.get_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( ): string
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

voltage→**get_logicalName()**

YVoltage

voltage→**logicalName()****voltage.get_logicalName()**

Returns the logical name of the voltage sensor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the voltage sensor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

voltage→**get_lowestValue()****YVoltage****voltage**→**lowestValue()****voltage.get_lowestValue()**

Returns the minimal value observed for the voltage.

```
function get_lowestValue( ): double
```

Returns :

a floating point number corresponding to the minimal value observed for the voltage

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

voltage→**get_module()**

YVoltage

voltage→**module()****voltage.get_module()**

Gets the YModule object for the device on which the function is located.

function **get_module**(): TYModule

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

voltage→**get_recordedData()****YVoltage****voltage**→**recordedData()****voltage.get_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime: int64, endTime: int64): TYDataSet
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voltage→get_reportFrequency()

YVoltage

voltage→reportFrequency()

voltage.get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( ): string
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

voltage→**get_resolution()****YVoltage****voltage**→**resolution()****voltage.get_resolution()**

Returns the resolution of the measured values.

```
function get_resolution( ): double
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

voltage→**get_unit()**

YVoltage

voltage→**unit()****voltage.get_unit()**

Returns the measuring unit for the voltage.

```
function get_unit( ): string
```

Returns :

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voltage→**get_userdata()****YVoltage****voltage**→**userData()****voltage.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

function **get_userdata()**: Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

voltage→isOnline()**voltage.isOnline()**

YVoltage

Checks if the voltage sensor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

Returns :

`true` if the voltage sensor can be reached, and `false` otherwise

voltage→load()voltage.load()**YVoltage**

Preloads the voltage sensor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

voltage→loadCalibrationPoints()**YVoltage****voltage.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function loadCalibrationPoints( var rawValues: TDoubleArray,  
                               var refValues: TDoubleArray): LongInt
```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**nextVoltage()****voltage.nextVoltage()****YVoltage**

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

```
function nextVoltage( ): TYVoltage
```

Returns :

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a `null` pointer if there are no more voltage sensors to enumerate.

voltage→registerTimedReportCallback()
voltage.registerTimedReportCallback()**YVoltage**

Registers the callback function that is invoked on every periodic timed notification.

```
function registerTimedReportCallback( callback: TYVoltageTimedReportCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

voltage→registerValueCallback()
voltage.registerValueCallback()

YVoltage

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYVoltageValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voltage→**set_highestValue()**

YVoltage

voltage→**setHighestValue()**

voltage.set_highestValue()

Changes the recorded maximal value observed pour the voltage.

```
function set_highestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed pour the voltage

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_logFrequency()****YVoltage****voltage**→**setLogFrequency()****voltage.set_logFrequency()**

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_logicalName()**

YVoltage

voltage→**setLogicalName()****voltage.set_logicalName()**

Changes the logical name of the voltage sensor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage sensor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

voltage→**set_lowestValue()****YVoltage****voltage**→**setLowestValue()****voltage.set_lowestValue()**

Changes the recorded minimal value observed pour the voltage.

```
function set_lowestValue( newval: double): integer
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed pour the voltage

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_reportFrequency()****YVoltage****voltage**→**setReportFrequency()****voltage.set_reportFrequency()**

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval: string): integer
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_resolution()****YVoltage****voltage**→**setResolution()****voltage.set_resolution()**

Changes the resolution of the measured values.

```
function set_resolution( newval: double): integer
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_userdata()**

YVoltage

voltage→**setUserData()****voltage.set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.42. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

js	<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)

3. Reference

	Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<code>vsource→get_overCurrent()</code>	Returns true if the appliance connected to the device is too greedy .
<code>vsource→get_overHeat()</code>	Returns TRUE if the module is overheating.
<code>vsource→get_overLoad()</code>	Returns true if the device is not able to maintain the requested voltage output .
<code>vsource→get_regulationFailure()</code>	Returns true if the voltage output is too high regarding the requested voltage .
<code>vsource→get_unit()</code>	Returns the measuring unit for the voltage.
<code>vsource→get_userData()</code>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<code>vsource→get_voltage()</code>	Returns the voltage output command (mV)
<code>vsource→isOnline()</code>	Checks if the function is currently reachable, without raising any error.
<code>vsource→isOnline_async(callback, context)</code>	Checks if the function is currently reachable, without raising any error (asynchronous version).
<code>vsource→load(msValidity)</code>	Preloads the function cache with a specified validity duration.
<code>vsource→load_async(msValidity, callback, context)</code>	Preloads the function cache with a specified validity duration (asynchronous version).
<code>vsource→nextVSource()</code>	Continues the enumeration of voltage sources started using <code>yFirstVSource()</code> .
<code>vsource→pulse(voltage, ms_duration)</code>	Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.
<code>vsource→registerValueCallback(callback)</code>	Registers the callback function that is invoked on every change of advertised value.
<code>vsource→set_logicalName(newval)</code>	Changes the logical name of the voltage source.
<code>vsource→set_userData(data)</code>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<code>vsource→set_voltage(newval)</code>	Tunes the device output voltage (milliVolts).
<code>vsource→voltageMove(target, ms_duration)</code>	Performs a smooth move at constant speed toward a given value.
<code>vsource→wait_async(callback, context)</code>	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

yFindVSource() — YVSource.FindVSource()yFindVSource()

YVSource

Retrieves a voltage source for a given identifier.

```
function yFindVSource( func: string): TYVSource
```

yFindVSource() — YVSource.FindVSource()yFindVSource()

Retrieves a voltage source for a given identifier.

js	function yFindVSource(func)
php	function yFindVSource(\$func)
cpp	YVSource* yFindVSource(const string& func)
m	YVSource* yFindVSource(NSString* func)
pas	function yFindVSource(func: string): TYVSource
vb	function yFindVSource(ByVal func As String) As YVSource
cs	YVSource FindVSource(string func)
java	YVSource FindVSource(String func)
py	def FindVSource(func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage source

Returns :

a YVSource object allowing you to drive the voltage source.

yFirstVSource() —

YVSource

YVSource.FirstVSource()yFirstVSource()

Starts the enumeration of voltage sources currently accessible.

```
function yFirstVSource( ): TYVSource
```

yFirstVSource() —

YVSource.FirstVSource()yFirstVSource()

Starts the enumeration of voltage sources currently accessible.

js	function yFirstVSource()
php	function yFirstVSource()
cpp	YVSource* yFirstVSource()
m	YVSource* yFirstVSource()
pas	function yFirstVSource(): TYVSource
vb	function yFirstVSource() As YVSource
cs	YVSource FirstVSource()
java	YVSource FirstVSource()
py	def FirstVSource()

Use the method `YVSource.nextVSource()` to iterate on next voltage sources.

Returns :
a pointer to a `YVSource` object, corresponding to the first voltage source currently online, or a `null` pointer if there are none.

vsource→describe()**vsource.describe()****YVSource**

Returns a short text that describes the function in the form `TYPE (NAME) =SERIAL . FUNCTIONID`.

```
function describe( ): string
```

vsource→describe()**vsource.describe()**

Returns a short text that describes the function in the form `TYPE (NAME) =SERIAL . FUNCTIONID`.

js	function describe ()
php	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	function describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the function (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

vsource→**get_advertisedValue()****YVSource****vsource**→**advertisedValue()****vsource.get_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

```
function get_advertisedValue( ): string
```

vsource→**get_advertisedValue()****vsource**→**advertisedValue()****vsource.get_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

js	function get_advertisedValue ()
php	function get_advertisedValue ()
cpp	string get_advertisedValue ()
m	-(NSString*) advertisedValue
pas	function get_advertisedValue (): string
vb	function get_advertisedValue () As String
cs	string get_advertisedValue ()
java	String get_advertisedValue ()
py	def get_advertisedValue ()
cmd	YVSource target get_advertisedValue

Returns :

a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

vsourcesrc→get_errorMessage()**YVSource****vsourcesrc→errorMessage()****vsourcesrc.get_errorMessage()**

Returns the error message of the latest error with this function.

```
function get_errorMessage( ): string
```

vsourcesrc→get_errorMessage()**vsourcesrc→errorMessage()vsourcesrc.get_errorMessage()**

Returns the error message of the latest error with this function.

<code>js</code>	<code>function get_errorMessage()</code>
<code>php</code>	<code>function get_errorMessage()</code>
<code>cpp</code>	<code>string get_errorMessage()</code>
<code>m</code>	<code>-(NSString*) errorMessage</code>
<code>pas</code>	<code>function get_errorMessage(): string</code>
<code>vb</code>	<code>function get_errorMessage() As String</code>
<code>cs</code>	<code>string get_errorMessage()</code>
<code>java</code>	<code>String get_errorMessage()</code>
<code>py</code>	<code>def get_errorMessage()</code>

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using this function object

vsource→**get_errorType()****YVSource****vsource**→**errorType()****vsource.get_errorType()**

Returns the numerical error code of the latest error with this function.

```
function get_errorType( ): YRETCODE
```

vsource→**get_errorType()****vsource**→**errorType()****vsource.get_errorType()**

Returns the numerical error code of the latest error with this function.

js	function get_errorType ()
php	function get_errorType ()
cpp	YRETCODE get_errorType ()
pas	function get_errorType (): YRETCODE
vb	function get_errorType () As YRETCODE
cs	YRETCODE get_errorType ()
java	int get_errorType ()
py	def get_errorType ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using this function object

vsources→get_extPowerFailure()**YVSource****vsources→extPowerFailure()****vsources.get_extPowerFailure()**

Returns true if external power supply voltage is too low.

function **get_extPowerFailure**(): Integer**vsources→get_extPowerFailure()****vsources→extPowerFailure(vsources.get_extPowerFailure())**

Returns true if external power supply voltage is too low.

js	function get_extPowerFailure ()
php	function get_extPowerFailure ()
cpp	Y_EXTPOWERFAILURE_enum get_extPowerFailure ()
m	-(Y_EXTPOWERFAILURE_enum) extPowerFailure
pas	function get_extPowerFailure (): Integer
vb	function get_extPowerFailure () As Integer
cs	int get_extPowerFailure ()
java	int get_extPowerFailure ()
py	def get_extPowerFailure ()
cmd	YVSource target get_extPowerFailure

Returns :

either Y_EXTPOWERFAILURE_FALSE or Y_EXTPOWERFAILURE_TRUE, according to true if external power supply voltage is too low

On failure, throws an exception or returns Y_EXTPOWERFAILURE_INVALID.

vsource→**get_failure()****YVSource****vsource**→**failure()****vsource.get_failure()**

Returns true if the module is in failure mode.

function **get_failure**(): Integer**vsource**→**get_failure()****vsource**→**failure()****vsource.get_failure()**

Returns true if the module is in failure mode.

js	function get_failure ()
php	function get_failure ()
cpp	Y_FAILURE_enum get_failure ()
m	-(Y_FAILURE_enum) failure
pas	function get_failure (): Integer
vb	function get_failure () As Integer
cs	int get_failure ()
java	int get_failure ()
py	def get_failure ()
cmd	YVSource target get_failure

More information can be obtained by testing `get_overheat`, `get_overcurrent` etc... When a error condition is met, the output voltage is set to zéro and cannot be changed until the `reset()` function is called.

Returns :

either `Y_FAILURE_FALSE` or `Y_FAILURE_TRUE`, according to true if the module is in failure mode

On failure, throws an exception or returns `Y_FAILURE_INVALID`.

vsource→**get_functionDescriptor()****YVSource****vsource**→**functionDescriptor()****vsource.get_vsourceDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

vsource→**get_functionDescriptor()****vsource**→**functionDescriptor()****vsource.get_vsourceDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor ()
php	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor (): YFUN_DESCR
vb	function get_functionDescriptor () As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor ()
java	String get_functionDescriptor ()
py	def get_functionDescriptor ()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

vsource→**get_logicalName()****YVSource****vsource**→**logicalName()****vsource.get_logicalName()**

Returns the logical name of the voltage source.

```
function get_logicalName( ): string
```

vsource→**get_logicalName()****vsource**→**logicalName()****vsource.get_logicalName()**

Returns the logical name of the voltage source.

js	function get_logicalName ()
php	function get_logicalName ()
cpp	string get_logicalName ()
m	-(NSString*) logicalName
pas	function get_logicalName (): string
vb	function get_logicalName () As String
cs	string get_logicalName ()
java	String get_logicalName ()
py	def get_logicalName ()
cmd	YVSource target get_logicalName

Returns :

a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

vsourcesource→get_module()**YVSource****vsourcesource→module()vsourcesource.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

vsourcesource→get_module()**vsourcesource→module()vsourcesource.get_module()**

Gets the YModule object for the device on which the function is located.

js	function get_module ()
php	function get_module ()
cpp	YModule * get_module ()
m	-(YModule*) module
pas	function get_module (): TModule
vb	function get_module () As YModule
cs	YModule get_module ()
java	YModule get_module ()
py	def get_module ()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

vsSource→**get_overCurrent()****YVSource****vsSource**→**overCurrent()****vsSource.get_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

```
function get_overCurrent( ): Integer
```

vsSource→**get_overCurrent()****vsSource**→**overCurrent()****vsSource.get_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

<code>js</code>	<code>function get_overCurrent()</code>
<code>php</code>	<code>function get_overCurrent()</code>
<code>cpp</code>	<code>Y_OVERCURRENT_enum get_overCurrent()</code>
<code>m</code>	<code>-(Y_OVERCURRENT_enum) overCurrent</code>
<code>pas</code>	<code>function get_overCurrent(): Integer</code>
<code>vb</code>	<code>function get_overCurrent() As Integer</code>
<code>cs</code>	<code>int get_overCurrent()</code>
<code>java</code>	<code>int get_overCurrent()</code>
<code>py</code>	<code>def get_overCurrent()</code>
<code>cmd</code>	<code>YVSource target get_overCurrent</code>

Returns :

either Y_OVERCURRENT_FALSE or Y_OVERCURRENT_TRUE, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns Y_OVERCURRENT_INVALID.

vsource→**get_overHeat()****YVSource****vsource**→**overHeat()****vsource.get_overHeat()**

Returns TRUE if the module is overheating.

function **get_overHeat**(): Integer**vsource**→**get_overHeat()****vsource**→**overHeat()****vsource.get_overHeat()**

Returns TRUE if the module is overheating.

js	function get_overHeat ()
php	function get_overHeat ()
cpp	Y_OVERHEAT_enum get_overHeat ()
m	-(Y_OVERHEAT_enum) overHeat
pas	function get_overHeat (): Integer
vb	function get_overHeat () As Integer
cs	int get_overHeat ()
java	int get_overHeat ()
py	def get_overHeat ()
cmd	YVSource target get_overHeat

Returns :

either Y_OVERHEAT_FALSE or Y_OVERHEAT_TRUE, according to TRUE if the module is overheating

On failure, throws an exception or returns Y_OVERHEAT_INVALID.

vsources→get_overLoad()**YVSource****vsources→overLoad()vsources.get_overLoad()**

Returns true if the device is not able to maintain the requested voltage output .

```
function get_overLoad( ): Integer
```

vsources→get_overLoad()**vsources→overLoad()vsources.get_overLoad()**

Returns true if the device is not able to maintain the requested voltage output .

<code>js</code>	<code>function get_overLoad()</code>
<code>php</code>	<code>function get_overLoad()</code>
<code>cpp</code>	<code>Y_OVERLOAD_enum get_overLoad()</code>
<code>m</code>	<code>-(Y_OVERLOAD_enum) overLoad</code>
<code>pas</code>	<code>function get_overLoad(): Integer</code>
<code>vb</code>	<code>function get_overLoad() As Integer</code>
<code>cs</code>	<code>int get_overLoad()</code>
<code>java</code>	<code>int get_overLoad()</code>
<code>py</code>	<code>def get_overLoad()</code>
<code>cmd</code>	<code>YVSource target get_overLoad</code>

Returns :

either Y_OVERLOAD_FALSE or Y_OVERLOAD_TRUE, according to true if the device is not able to maintain the requested voltage output

On failure, throws an exception or returns Y_OVERLOAD_INVALID.

vsources→get_regulationFailure()**YVSource****vsources→regulationFailure()****vsources.get_regulationFailure()**

Returns true if the voltage output is too high regarding the requested voltage .

```
function get_regulationFailure( ): Integer
```

vsources→get_regulationFailure()**vsources→regulationFailure()****vsources.get_regulationFailure()**

Returns true if the voltage output is too high regarding the requested voltage .

js	function get_regulationFailure ()
php	function get_regulationFailure ()
cpp	Y_REGULATIONFAILURE_enum get_regulationFailure ()
m	-(Y_REGULATIONFAILURE_enum) regulationFailure
pas	function get_regulationFailure (): Integer
vb	function get_regulationFailure () As Integer
cs	int get_regulationFailure ()
java	int get_regulationFailure ()
py	def get_regulationFailure ()
cmd	YVSource target get_regulationFailure

Returns :

either Y_REGULATIONFAILURE_FALSE or Y_REGULATIONFAILURE_TRUE, according to true if the voltage output is too high regarding the requested voltage

On failure, throws an exception or returns Y_REGULATIONFAILURE_INVALID.

vsource→**get_unit()**
vsource→**unit()****vsource.get_unit()**

YVSource

Returns the measuring unit for the voltage.

```
function get_unit( ): string
```

vsource→**get_unit()**
vsource→**unit()****vsource.get_unit()**

Returns the measuring unit for the voltage.

js	function get_unit ()
php	function get_unit ()
cpp	string get_unit ()
m	-(NSString*) unit
pas	function get_unit (): string
vb	function get_unit () As String
cs	string get_unit ()
java	String get_unit ()
py	def get_unit ()
cmd	YVSource target get_unit

Returns :
a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns Y_UNIT_INVALID.

vsource→**get_userData()****YVSource****vsource**→**userData()****vsource.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

vsource→**get_userData()****vsource**→**userData()****vsource.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	function get_userData ()
php	function get_userData ()
cpp	void * get_userData ()
m	-(void*) userData
pas	function get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	def get_userData ()

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

vssource→**get_voltage()**
vssource→**voltage()****vs**source.**get_voltage()**

YVSource

Returns the voltage output command (mV)

```
function get_voltage( ): LongInt
```

vssource→**get_voltage()**
vssource→**voltage()****vs**source.**get_voltage()**

Returns the voltage output command (mV)

js	function get_voltage ()
php	function get_voltage ()
cpp	int get_voltage ()
m	-(int) voltage
pas	function get_voltage (): LongInt
vb	function get_voltage () As Integer
cs	int get_voltage ()
java	int get_voltage ()
py	def get_voltage ()

Returns :
an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns Y_VOLTAGE_INVALID.

vsource→isOnline()**vsource.isOnline()****YVSource**

Checks if the function is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

vsource→isOnline()**vsource.isOnline()**

Checks if the function is currently reachable, without raising any error.

js	function isOnline ()
php	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	function isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	def isOnline ()

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

Returns :

`true` if the function can be reached, and `false` otherwise

vsource→load()**vsource.load()****YVSource**

Preloads the function cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

vsource→load()**vsource.load()**

Preloads the function cache with a specified validity duration.

js	function load(msValidity)
php	function load(\$msValidity)
cpp	YRETCODE load(int msValidity)
m	-(YRETCODE) load : (int) msValidity
pas	function load(msValidity: integer): YRETCODE
vb	function load(ByVal msValidity As Integer) As YRETCODE
cs	YRETCODE load(int msValidity)
java	int load(long msValidity)
py	def load(msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

vsources→nextVSource()vsources.nextVSource()**YVSource**

Continues the enumeration of voltage sources started using `yFirstVSource()`.

```
function nextVSource( ): TYVSource
```

vsources→nextVSource()vsources.nextVSource()

Continues the enumeration of voltage sources started using `yFirstVSource()`.

js	function nextVSource()
php	function nextVSource()
cpp	YVSource * nextVSource()
m	-(YVSource*) nextVSource
pas	function nextVSource(): TYVSource
vb	function nextVSource() As YVSource
cs	YVSource nextVSource()
java	YVSource nextVSource()
py	def nextVSource()

Returns :

a pointer to a `YVSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more voltage sources to enumerate.

vsource→pulse()vsource.pulse()**YVSource**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

```
function pulse( voltage: integer, ms_duration: integer): integer
```

vsource→pulse()vsource.pulse()

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

js	function pulse (voltage , ms_duration)
php	function pulse (\$ voltage , \$ ms_duration)
cpp	int pulse (int voltage , int ms_duration)
m	-(int) pulse : (int) voltage : (int) ms_duration
pas	function pulse (voltage : integer, ms_duration : integer): integer
vb	function pulse (ByVal voltage As Integer, ByVal ms_duration As Integer) As Integer
cs	int pulse (int voltage , int ms_duration)
java	int pulse (int voltage , int ms_duration)
py	def pulse (voltage , ms_duration)
cmd	YVSource target pulse voltage ms_duration

Parameters :

voltage pulse voltage, in millivolts
ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→registerValueCallback() **vsource.registerValueCallback()**

YVSource

Registers the callback function that is invoked on every change of advertised value.

```
procedure registerValueCallback( callback: TGenericUpdateCallback)
```

vsource→registerValueCallback()**vsource.registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

js	function registerValueCallback (callback)
php	function registerValueCallback (\$callback)
cpp	void registerValueCallback (YDisplayUpdateCallback callback)
pas	procedure registerValueCallback (callback : TGenericUpdateCallback)
vb	procedure registerValueCallback (ByVal callback As GenericUpdateCallback)
cs	void registerValueCallback (UpdateCallback callback)
java	void registerValueCallback (UpdateCallback callback)
py	def registerValueCallback (callback)
m	-(void) registerValueCallback : (YFunctionUpdateCallback) callback

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

vsource→**set_logicalName()****YVSource****vsource**→**setLogicalName()****vsource.set_logicalName()**

Changes the logical name of the voltage source.

```
function set_logicalName( newval: string): integer
```

vsource→**set_logicalName()****vsource**→**setLogicalName()****vsource.set_logicalName()**

Changes the logical name of the voltage source.

js	function set_logicalName(newval)
php	function set_logicalName(\$newval)
cpp	int set_logicalName(const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function set_logicalName(newval: string): integer
vb	function set_logicalName(ByVal newval As String) As Integer
cs	int set_logicalName(string newval)
java	int set_logicalName(String newval)
py	def set_logicalName(newval)
cmd	YVSource target set_logicalName newval

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage source

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→**set_userData()****YVSource****vsource**→**setUserData()****vsource.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userData( data: Tobject)
```

vsource→**set_userData()****vsource**→**setUserData()****vsource.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

js	function set_userData (data)
php	function set_userData (\$data)
cpp	void set_userData (void* data)
m	-(void) setUserData : (void*) data
pas	procedure set_userData (data : Tobject)
vb	procedure set_userData (ByVal data As Object)
cs	void set_userData (object data)
java	void set_userData (Object data)
py	def set_userData (data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

vsource→**set_voltage()****YVSource****vsource**→**setVoltage()****vsource.set_voltage()**

Tunes the device output voltage (milliVolts).

```
function set_voltage( newval: LongInt): integer
```

vsource→**set_voltage()****vsource**→**setVoltage()****vsource.set_voltage()**

Tunes the device output voltage (milliVolts).

js	function set_voltage (newval)
php	function set_voltage (\$newval)
cpp	int set_voltage (int newval)
m	-(int) setVoltage : (int) newval
pas	function set_voltage (newval : LongInt): integer
vb	function set_voltage (ByVal newval As Integer) As Integer
cs	int set_voltage (int newval)
java	int set_voltage (int newval)
py	def set_voltage (newval)
cmd	YVSource target set_voltage newval

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→voltageMove()**vsource.voltageMove()****YVSource**

Performs a smooth move at constant speed toward a given value.

```
function voltageMove( target: integer, ms_duration: integer): integer
```

vsource→voltageMove()**vsource.voltageMove()**

Performs a smooth move at constant speed toward a given value.

<code>js</code>	<code>function voltageMove(target, ms_duration)</code>
<code>php</code>	<code>function voltageMove(\$target, \$ms_duration)</code>
<code>cpp</code>	<code>int voltageMove(int target, int ms_duration)</code>
<code>m</code>	<code>-(int) voltageMove : (int) target : (int) ms_duration</code>
<code>pas</code>	<code>function voltageMove(target: integer, ms_duration: integer): integer</code>
<code>vb</code>	<code>function voltageMove(ByVal target As Integer, ByVal ms_duration As Integer) As Integer</code>
<code>cs</code>	<code>int voltageMove(int target, int ms_duration)</code>
<code>java</code>	<code>int voltageMove(int target, int ms_duration)</code>
<code>py</code>	<code>def voltageMove(target, ms_duration)</code>
<code>cmd</code>	<code>YVSource target voltageMove target ms_duration</code>

Parameters :

target new output value at end of transition, in milliVolts.
ms_duration transition duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.43. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
php	require_once('yocto_wakeupmonitor.php');
c++	#include "yocto_wakeupmonitor.h"
m	#import "yocto_wakeupmonitor.h"
pas	uses yocto_wakeupmonitor;
vb	yocto_wakeupmonitor.vb
cs	yocto_wakeupmonitor.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py	from yocto_wakeupmonitor import *

Global functions
yFindWakeUpMonitor(func) Retrieves a monitor for a given identifier.
yFirstWakeUpMonitor() Starts the enumeration of monitors currently accessible.
YWakeupMonitor methods
wakeupmonitor→describe() Returns a short text that describes unambiguously the instance of the monitor in the form TYPE (NAME) =SERIAL . FUNCTIONID.
wakeupmonitor→get_advertisedValue() Returns the current value of the monitor (no more than 6 characters).
wakeupmonitor→get_errorMessage() Returns the error message of the latest error with the monitor.
wakeupmonitor→get_errorType() Returns the numerical error code of the latest error with the monitor.
wakeupmonitor→get_friendlyName() Returns a global identifier of the monitor in the format MODULE_NAME . FUNCTION_NAME.
wakeupmonitor→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
wakeupmonitor→get_functionId() Returns the hardware identifier of the monitor, without reference to the module.
wakeupmonitor→get_hardwareId() Returns the unique hardware identifier of the monitor in the form SERIAL . FUNCTIONID.
wakeupmonitor→get_logicalName() Returns the logical name of the monitor.
wakeupmonitor→get_module() Gets the YModule object for the device on which the function is located.
wakeupmonitor→get_module_async(callback, context) Gets the YModule object for the device on which the function is located (asynchronous version).
wakeupmonitor→get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)
wakeupmonitor→get_powerDuration() Returns the maximal wake up time (in seconds) before automatically going to sleep.
wakeupmonitor→get_sleepCountdown() Returns the delay before the next sleep period.
wakeupmonitor→get_userData() Returns the value of the userData attribute, as previously stored using method set_userData.
wakeupmonitor→get_wakeUpReason() Returns the latest wake up reason.
wakeupmonitor→get_wakeUpState() Returns the current state of the monitor
wakeupmonitor→isOnline() Checks if the monitor is currently reachable, without raising any error.
wakeupmonitor→isOnline_async(callback, context) Checks if the monitor is currently reachable, without raising any error (asynchronous version).
wakeupmonitor→load(msValidity) Preloads the monitor cache with a specified validity duration.
wakeupmonitor→load_async(msValidity, callback, context) Preloads the monitor cache with a specified validity duration (asynchronous version).
wakeupmonitor→nextWakeUpMonitor() Continues the enumeration of monitors started using yFirstWakeUpMonitor().
wakeupmonitor→registerValueCallback(callback) Registers the callback function that is invoked on every change of advertised value.
wakeupmonitor→resetSleepCountDown() Resets the sleep countdown.
wakeupmonitor→set_logicalName(newval) Changes the logical name of the monitor.
wakeupmonitor→set_nextWakeUp(newval) Changes the days of the week when a wake up must take place.
wakeupmonitor→set_powerDuration(newval) Changes the maximal wake up time (seconds) before automatically going to sleep.
wakeupmonitor→set_sleepCountdown(newval) Changes the delay before the next sleep period.
wakeupmonitor→set_userData(data) Stores a user context provided as argument in the userData attribute of the function.
wakeupmonitor→sleep(secBeforeSleep) Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.
wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep) Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.
wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep) Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.
wakeupmonitor→wait_async(callback, context)

3. Reference

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wakeupmonitor→**wakeUp()**

Forces a wake up.

YWakeUpMonitor.FindWakeUpMonitor() yFindWakeUpMonitor()yFindWakeUpMonitor()

YWakeupMonitor

Retrieves a monitor for a given identifier.

```
function yFindWakeUpMonitor( func: string): TYWakeUpMonitor
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeupMonitor.isOnline()` to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the monitor

Returns :

a `YWakeupMonitor` object allowing you to drive the monitor.

YWakeUpMonitor.FirstWakeUpMonitor()

YWakeUpMonitor

yFirstWakeUpMonitor()**yFirstWakeUpMonitor()**

Starts the enumeration of monitors currently accessible.

```
function yFirstWakeUpMonitor( ): TYWakeUpMonitor
```

Use the method `YWakeUpMonitor.nextWakeUpMonitor()` to iterate on next monitors.

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to the first monitor currently online, or a `null` pointer if there are none.

wakeupmonitor→describe()
wakeupmonitor.describe()

YWakeUpMonitor

Returns a short text that describes unambiguously the instance of the monitor in the form
`TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the monitor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupmonitor→get_advertisedValue()

YWakeUpMonitor

wakeupmonitor→advertisedValue()

wakeupmonitor.get_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the monitor (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wakeupmonitor→get_errorMessage()**YWakeUpMonitor****wakeupmonitor→errorMessage()****wakeupmonitor.get_errorMessage()**

Returns the error message of the latest error with the monitor.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the monitor object

wakeupmonitor→get_errorType()

YWakeUpMonitor

wakeupmonitor→errorType()

wakeupmonitor.get_errorType()

Returns the numerical error code of the latest error with the monitor.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the monitor object

wakeupmonitor→**get_functionDescriptor()****YWakeUpMonitor****wakeupmonitor**→**functionDescriptor()****wakeupmonitor.get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupmonitor→get_logicalName()
wakeupmonitor→logicalName()
wakeupmonitor.get_logicalName()

YWakeUpMonitor

Returns the logical name of the monitor.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the monitor. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wakeupmonitor→**get_module()****YWakeUpMonitor****wakeupmonitor**→**module()****wakeupmonitor.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

wakeupmonitor→get_nextWakeUp()

YWakeUpMonitor

wakeupmonitor→nextWakeUp()

wakeupmonitor.get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)

```
function get_nextWakeUp( ): int64
```

Returns :

an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns Y_NEXTWAKEUP_INVALID.

wakeupmonitor→get_powerDuration()**YWakeUpMonitor****wakeupmonitor→powerDuration()****wakeupmonitor.get_powerDuration()**

Returns the maximal wake up time (in seconds) before automatically going to sleep.

```
function get_powerDuration( ): LongInt
```

Returns :

an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns Y_POWERDURATION_INVALID.

wakeupmonitor→get_sleepCountdown()

YWakeUpMonitor

wakeupmonitor→sleepCountdown()

wakeupmonitor.get_sleepCountdown()

Returns the delay before the next sleep period.

```
function get_sleepCountdown( ): LongInt
```

Returns :

an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns Y_SLEEPDOWNDOWN_INVALID.

wakeupmonitor→get_userdata()**YWakeUpMonitor****wakeupmonitor→userData()****wakeupmonitor.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
function get_userdata( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

wakeupmonitor→get_wakeUpReason()**YWakeUpMonitor****wakeupmonitor→wakeUpReason()****wakeupmonitor.get_wakeUpReason()**

Returns the latest wake up reason.

```
function get_wakeUpReason( ): Integer
```

Returns :

a value among Y_WAKEUPREASON_USBPOWER, Y_WAKEUPREASON_EXTPOWER, Y_WAKEUPREASON_ENDOFSLEEP, Y_WAKEUPREASON_EXTSIG1, Y_WAKEUPREASON_EXTSIG2, Y_WAKEUPREASON_EXTSIG3, Y_WAKEUPREASON_EXTSIG4, Y_WAKEUPREASON_SCHEDULE1, Y_WAKEUPREASON_SCHEDULE2, Y_WAKEUPREASON_SCHEDULE3, Y_WAKEUPREASON_SCHEDULE4, Y_WAKEUPREASON_SCHEDULE5 and Y_WAKEUPREASON_SCHEDULE6 corresponding to the latest wake up reason

On failure, throws an exception or returns Y_WAKEUPREASON_INVALID.

wakeupmonitor→**get_wakeUpState()****YWakeUpMonitor****wakeupmonitor**→**wakeUpState()****wakeupmonitor.get_wakeUpState()**

Returns the current state of the monitor

```
function get_wakeUpState( ): Integer
```

Returns :

either Y_WAKEUPSTATE_SLEEPING or Y_WAKEUPSTATE_AWAKE, according to the current state of the monitor

On failure, throws an exception or returns Y_WAKEUPSTATE_INVALID.

wakeupmonitor→isOnline()wakeupmonitor.isOnline()

YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

Returns :

`true` if the monitor can be reached, and `false` otherwise

wakeupmonitor→load()wakeupmonitor.load()**YWakeUpMonitor**

Preloads the monitor cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→nextWakeUpMonitor()
wakeupmonitor.nextWakeUpMonitor()

YWakeUpMonitor

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

```
function nextWakeUpMonitor( ): TYWakeUpMonitor
```

Returns :

a pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a `null` pointer if there are no more monitors to enumerate.

wakeupmonitor→registerValueCallback()
wakeupmonitor.registerValueCallback()

YWakeUpMonitor

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYWakeUpMonitorValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupmonitor→resetSleepCountDown()
wakeupmonitor.resetSleepCountDown()

YWakeUpMonitor

Resets the sleep countdown.

```
function resetSleepCountDown( ): LongInt
```

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→**set_logicalName()****YWakeUpMonitor****wakeupmonitor**→**setLogicalName()****wakeupmonitor.set_logicalName()**

Changes the logical name of the monitor.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the monitor.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_nextWakeUp()

YWakeUpMonitor

wakeupmonitor→setNextWakeUp()

wakeupmonitor.set_nextWakeUp()

Changes the days of the week when a wake up must take place.

```
function set_nextWakeUp( newval: int64): integer
```

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_powerDuration()**YWakeUpMonitor****wakeupmonitor→setPowerDuration()****wakeupmonitor.set_powerDuration()**

Changes the maximal wake up time (seconds) before automatically going to sleep.

```
function set_powerDuration( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_sleepCountdown()
wakeupmonitor→setSleepCountdown()
wakeupmonitor.set_sleepCountdown()

YWakeUpMonitor

Changes the delay before the next sleep period.

```
function set_sleepCountdown( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the delay before the next sleep period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_userData()**YWakeUpMonitor****wakeupmonitor→setUserData()****wakeupmonitor.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userData( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

wakeupmonitor→sleep()wakeupmonitor.sleep()**YWakeUpMonitor**

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

```
function sleep( secBeforeSleep: LongInt): LongInt
```

Parameters :

secBeforeSleep number of seconds before going into sleep mode,

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepFor()
wakeupmonitor.sleepFor()

YWakeUpMonitor

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
function sleepFor( secUntilWakeUp: LongInt,  
                  secBeforeSleep: LongInt): LongInt
```

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

secUntilWakeUp sleep duration, in secondes

secBeforeSleep number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepUntil() wakeupmonitor.sleepUntil()

YWakeUpMonitor

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
function sleepUntil( wakeUpTime: LongInt,  
                    secBeforeSleep: LongInt): LongInt
```

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

wakeUpTime wake-up datetime (UNIX format)
secBeforeSleep number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→**wakeUp()****wakeupmonitor.wakeUp()****YWakeUpMonitor**

Forces a wake up.

```
function wakeUp( ): LongInt
```

3.44. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupschedule.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWakeUpSchedule = yoctolib.YWakeUpSchedule;
php	require_once('yocto_wakeupschedule.php');
c++	#include "yocto_wakeupschedule.h"
m	#import "yocto_wakeupschedule.h"
pas	uses yocto_wakeupschedule;
vb	yocto_wakeupschedule.vb
cs	yocto_wakeupschedule.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
py	from yocto_wakeupschedule import *

Global functions
yFindWakeUpSchedule(func) Retrieves a wake up schedule for a given identifier.
yFirstWakeUpSchedule() Starts the enumeration of wake up schedules currently accessible.
YWakeUpSchedule methods
wakeupschedule→describe() Returns a short text that describes unambiguously the instance of the wake up schedule in the form TYPE (NAME) = SERIAL . FUNCTIONID.
wakeupschedule→get_advertisedValue() Returns the current value of the wake up schedule (no more than 6 characters).
wakeupschedule→get_errorMessage() Returns the error message of the latest error with the wake up schedule.
wakeupschedule→get_errorType() Returns the numerical error code of the latest error with the wake up schedule.
wakeupschedule→get_friendlyName() Returns a global identifier of the wake up schedule in the format MODULE_NAME . FUNCTION_NAME.
wakeupschedule→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
wakeupschedule→get_functionId() Returns the hardware identifier of the wake up schedule, without reference to the module.
wakeupschedule→get_hardwareId() Returns the unique hardware identifier of the wake up schedule in the form SERIAL . FUNCTIONID.
wakeupschedule→get_hours() Returns the hours scheduled for wake up.
wakeupschedule→get_logicalName() Returns the logical name of the wake up schedule.
wakeupschedule→get_minutes() Returns all the minutes of each hour that are scheduled for wake up.
wakeupschedule→get_minutesA()

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

wakeupschedule→get_minutesB()

Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

wakeupschedule→get_module()

Gets the YModule object for the device on which the function is located.

wakeupschedule→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

wakeupschedule→get_monthDays()

Returns the days of the month scheduled for wake up.

wakeupschedule→get_months()

Returns the months scheduled for wake up.

wakeupschedule→get_nextOccurence()

Returns the date/time (seconds) of the next wake up occurrence

wakeupschedule→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

wakeupschedule→get_weekDays()

Returns the days of the week scheduled for wake up.

wakeupschedule→isOnline()

Checks if the wake up schedule is currently reachable, without raising any error.

wakeupschedule→isOnline_async(callback, context)

Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

wakeupschedule→load(msValidity)

Preloads the wake up schedule cache with a specified validity duration.

wakeupschedule→load_async(msValidity, callback, context)

Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

wakeupschedule→nextWakeUpSchedule()

Continues the enumeration of wake up schedules started using yFirstWakeUpSchedule().

wakeupschedule→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

wakeupschedule→set_hours(newval)

Changes the hours when a wake up must take place.

wakeupschedule→set_logicalName(newval)

Changes the logical name of the wake up schedule.

wakeupschedule→set_minutes(bitmap)

Changes all the minutes where a wake up must take place.

wakeupschedule→set_minutesA(newval)

Changes the minutes in the 00-29 interval when a wake up must take place.

wakeupschedule→set_minutesB(newval)

Changes the minutes in the 30-59 interval when a wake up must take place.

wakeupschedule→set_monthDays(newval)

Changes the days of the month when a wake up must take place.

wakeupschedule→set_months(newval)

Changes the months when a wake up must take place.

wakeupschedule→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

3. Reference

wakeupschedule→**set_weekDays**(newval)

Changes the days of the week when a wake up must take place.

wakeupschedule→**wait_async**(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWakeUpSchedule.FindWakeUpSchedule() yFindWakeUpSchedule()yFindWakeUpSchedule()

YWakeUpSchedule

Retrieves a wake up schedule for a given identifier.

```
function yFindWakeUpSchedule( func: string): TYWakeUpSchedule
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpSchedule.isOnline()` to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the wake up schedule

Returns :

a `YWakeUpSchedule` object allowing you to drive the wake up schedule.

YWakeUpSchedule.FirstWakeUpSchedule() yFirstWakeUpSchedule()yFirstWakeUpSchedule()

YWakeUpSchedule

Starts the enumeration of wake up schedules currently accessible.

```
function yFirstWakeUpSchedule( ): TYWakeUpSchedule
```

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to the first wake up schedule currently online,
or a `null` pointer if there are none.

wakeupschedule→describe()
wakeupschedule.describe()

YWakeUpSchedule

Returns a short text that describes unambiguously the instance of the wake up schedule in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the wake up schedule (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupschedule→get_advertisedValue()

YWakeUpSchedule

wakeupschedule→advertisedValue()

wakeupschedule.get_advertisedValue()

Returns the current value of the wake up schedule (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the wake up schedule (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wakeupschedule→get_errorMessage()**YWakeUpSchedule****wakeupschedule→errorMessage()****wakeupschedule.get_errorMessage()**

Returns the error message of the latest error with the wake up schedule.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the wake up schedule object

wakeupschedule→get_errorType()

YWakeUpSchedule

wakeupschedule→errorType()

wakeupschedule.get_errorType()

Returns the numerical error code of the latest error with the wake up schedule.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the wake up schedule object

wakeupschedule→get_functionDescriptor()**YWakeUpSchedule****wakeupschedule→functionDescriptor()****wakeupschedule.get_functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wakeupschedule→get_hours()

YWakeUpSchedule

wakeupschedule→hours()

wakeupschedule.get_hours()

Returns the hours scheduled for wake up.

```
function get_hours( ): LongInt
```

Returns :

an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns Y_HOURS_INVALID.

wakeupschedule→get_logicalName()**YWakeUpSchedule****wakeupschedule→logicalName()****wakeupschedule.get_logicalName()**

Returns the logical name of the wake up schedule.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the wake up schedule. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wakeupschedule→get_minutes()

YWakeUpSchedule

wakeupschedule→minutes()

wakeupschedule.get_minutes()

Returns all the minutes of each hour that are scheduled for wake up.

```
function get_minutes( ): int64
```

wakeupschedule→get_minutesA()**YWakeUpSchedule****wakeupschedule→minutesA()****wakeupschedule.get_minutesA()**

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

```
function get_minutesA( ): LongInt
```

Returns :

an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y_MINUTESA_INVALID.

wakeupschedule→get_minutesB()

YWakeUpSchedule

wakeupschedule→minutesB()

wakeupschedule.get_minutesB()

Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

```
function get_minutesB( ): LongInt
```

Returns :

an integer corresponding to the minutes in the 30-59 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y_MINUTESB_INVALID.

wakeupschedule→get_module()**YWakeUpSchedule****wakeupschedule→module()****wakeupschedule.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

wakeupschedule→get_monthDays()

YWakeUpSchedule

wakeupschedule→monthDays()

wakeupschedule.get_monthDays()

Returns the days of the month scheduled for wake up.

```
function get_monthDays( ): LongInt
```

Returns :

an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns Y_MONTHDAYS_INVALID.

wakeupschedule→get_months()**YWakeUpSchedule****wakeupschedule→months()****wakeupschedule.get_months()**

Returns the months scheduled for wake up.

```
function get_months( ): LongInt
```

Returns :

an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns Y_MONTHS_INVALID.

wakeupschedule→get_nextOccurence()

YWakeUpSchedule

wakeupschedule→nextOccurence()

wakeupschedule.get_nextOccurence()

Returns the date/time (seconds) of the next wake up occurrence

```
function get_nextOccurence( ): int64
```

Returns :

an integer corresponding to the date/time (seconds) of the next wake up occurrence

On failure, throws an exception or returns Y_NEXTOCCURENCE_INVALID.

wakeupschedule→get_userdata()**YWakeUpSchedule****wakeupschedule→userData()****wakeupschedule.get_userdata()**

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

```
function get_userdata( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

wakeupschedule→get_weekDays()

YWakeUpSchedule

wakeupschedule→weekDays()

wakeupschedule.get_weekDays()

Returns the days of the week scheduled for wake up.

```
function get_weekDays( ): LongInt
```

Returns :

an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns Y_WEEKDAYS_INVALID.

**wakeupschedule→isOnline()
wakeupschedule.isOnline()**

YWakeUpSchedule

Checks if the wake up schedule is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

Returns :

`true` if the wake up schedule can be reached, and `false` otherwise

wakeupschedule→**load()****wakeupschedule.load()****YWakeUpSchedule**

Preloads the wake up schedule cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupschedule→nextWakeUpSchedule()
wakeupschedule.nextWakeUpSchedule()**

YWakeUpSchedule

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

```
function nextWakeUpSchedule( ): TYWakeUpSchedule
```

Returns :

a pointer to a `YWakeUpSchedule` object, corresponding to a wake up schedule currently online, or a `null` pointer if there are no more wake up schedules to enumerate.

**wakeupschedule→registerValueCallback()
wakeupschedule.registerValueCallback()****YWakeUpSchedule**

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYWakeUpScheduleValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupschedule→set_hours()**YWakeUpSchedule****wakeupschedule→setHours()****wakeupschedule.set_hours()**

Changes the hours when a wake up must take place.

```
function set_hours( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the hours when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_logicalName()

YWakeUpSchedule

wakeupschedule→setLogicalName()

wakeupschedule.set_logicalName()

Changes the logical name of the wake up schedule.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wake up schedule.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutes()**YWakeUpSchedule****wakeupschedule→setMinutes()****wakeupschedule.set_minutes()**

Changes all the minutes where a wake up must take place.

```
function set_minutes( bitmap: int64): LongInt
```

Parameters :

bitmap Minutes 00-59 of each hour scheduled for wake up.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesA()

YWakeUpSchedule

wakeupschedule→setMinutesA()

wakeupschedule.set_minutesA()

Changes the minutes in the 00-29 interval when a wake up must take place.

```
function set_minutesA( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesB()**YWakeUpSchedule****wakeupschedule→setMinutesB()****wakeupschedule.set_minutesB()**

Changes the minutes in the 30-59 interval when a wake up must take place.

```
function set_minutesB( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_monthDays()

YWakeUpSchedule

wakeupschedule→setMonthDays()

wakeupschedule.set_monthDays()

Changes the days of the month when a wake up must take place.

```
function set_monthDays( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the days of the month when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_months()**YWakeUpSchedule****wakeupschedule→setMonths()****wakeupschedule.set_months()**

Changes the months when a wake up must take place.

```
function set_months( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the months when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_userdata()

YWakeUpSchedule

wakeupschedule→setUserData()

wakeupschedule.set_userdata()

Stores a user context provided as argument in the userData attribute of the function.

```
procedure set_userdata( data: Tobject)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

wakeupschedule→set_weekDays()**YWakeUpSchedule****wakeupschedule→setWeekDays()****wakeupschedule.set_weekDays()**

Changes the days of the week when a wake up must take place.

```
function set_weekDays( newval: LongInt): integer
```

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.45. Watchdog function interface

The watchdog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven directly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_watchdog.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWatchdog = yoctolib.YWatchdog;
php	require_once('yocto_watchdog.php');
c++	#include "yocto_watchdog.h"
m	#import "yocto_watchdog.h"
pas	uses yocto_watchdog;
vb	yocto_watchdog.vb
cs	yocto_watchdog.cs
java	import com.yoctopuce.YoctoAPI.YWatchdog;
py	from yocto_watchdog import *

Global functions

yFindWatchdog(func)

Retrieves a watchdog for a given identifier.

yFirstWatchdog()

Starts the enumeration of watchdog currently accessible.

YWatchdog methods

watchdog→delayedPulse(ms_delay, ms_duration)

Schedules a pulse.

watchdog→describe()

Returns a short text that describes unambiguously the instance of the watchdog in the form
TYPE (NAME) =SERIAL . FUNCTIONID.

watchdog→get_advertisedValue()

Returns the current value of the watchdog (no more than 6 characters).

watchdog→get_autoStart()

Returns the watchdog running state at module power on.

watchdog→get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

watchdog→get_errorMessage()

Returns the error message of the latest error with the watchdog.

watchdog→get_errorType()

Returns the numerical error code of the latest error with the watchdog.

watchdog→get_friendlyName()

Returns a global identifier of the watchdog in the format MODULE_NAME . FUNCTION_NAME.

watchdog→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

watchdog→get_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

watchdog→get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

watchdog→get_logicalName()

Returns the logical name of the watchdog.

watchdog→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state A before automatically switching back in to B state.

watchdog→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state B before automatically switching back in to A state.

watchdog→get_module()

Gets the `YModule` object for the device on which the function is located.

watchdog→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

watchdog→get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

watchdog→get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

watchdog→get_running()

Returns the watchdog running state.

watchdog→get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

watchdog→get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, `UNCHANGED` for no change).

watchdog→get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

watchdog→get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

watchdog→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

watchdog→isOnline()

Checks if the watchdog is currently reachable, without raising any error.

watchdog→isOnline_async(callback, context)

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

watchdog→load(msValidity)

Preloads the watchdog cache with a specified validity duration.

watchdog→load_async(msValidity, callback, context)

Preloads the watchdog cache with a specified validity duration (asynchronous version).

watchdog→nextWatchdog()

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

watchdog→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

watchdog→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

watchdog→resetWatchdog()

Resets the watchdog.

watchdog→set_autoStart(newval)

Changes the watchdog running state at module power on.

watchdog→set_logicalName(newval)

Changes the logical name of the watchdog.

watchdog→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

watchdog→set_maxTimeOnStateB(newval)

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

watchdog→set_output(newval)

Changes the output state of the watchdog, when used as a simple switch (single throw).

watchdog→set_running(newval)

Changes the running state of the watchdog.

watchdog→set_state(newval)

Changes the state of the watchdog (A for the idle position, B for the active position).

watchdog→set_stateAtPowerOn(newval)

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

watchdog→set_triggerDelay(newval)

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

watchdog→set_triggerDuration(newval)

Changes the duration of resets caused by the watchdog, in milliseconds.

watchdog→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

watchdog→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWatchdog.FindWatchdog() yFindWatchdog()yFindWatchdog()

YWatchdog

Retrieves a watchdog for a given identifier.

```
function yFindWatchdog( func: string): TYWatchdog
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWatchdog.isOnline()` to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the watchdog

Returns :

a `YWatchdog` object allowing you to drive the watchdog.

YWatchdog.FirstWatchdog() yFirstWatchdog()yFirstWatchdog()

YWatchdog

Starts the enumeration of watchdog currently accessible.

```
function yFirstWatchdog( ): TYWatchdog
```

Use the method `YWatchdog.nextWatchdog()` to iterate on next watchdog.

Returns :

a pointer to a `YWatchdog` object, corresponding to the first watchdog currently online, or a `null` pointer if there are none.

watchdog→delayedPulse()watchdog.delayedPulse()

YWatchdog

Schedules a pulse.

```
function delayedPulse( ms_delay: LongInt, ms_duration: LongInt): integer
```

Parameters :

ms_delay waiting time before the pulse, in milliseconds

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→describe()**watchdog.describe()****YWatchdog**

Returns a short text that describes unambiguously the instance of the watchdog in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

function **describe**(): string

More precisely, `TYPE` is the type of the function, `NAME` is the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the watchdog (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

watchdog→get_advertisedValue()**YWatchdog****watchdog→advertisedValue()****watchdog.get_advertisedValue()**

Returns the current value of the watchdog (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the watchdog (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

watchdog→**get_autoStart()**

YWatchdog

watchdog→**autoStart()****watchdog.get_autoStart()**

Returns the watchdog runing state at module power on.

function **get_autoStart**(): Integer

Returns :

either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog runing state at module power on

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

watchdog→get_countdown()**YWatchdog****watchdog→countdown()watchdog.get_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

```
function get_countdown( ): int64
```

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns Y_COUNTDOWN_INVALID.

watchdog→get_errorMessage()

YWatchdog

watchdog→errorMessage()

watchdog.get_errorMessage()

Returns the error message of the latest error with the watchdog.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the watchdog object

watchdog→**get_errorType()****YWatchdog****watchdog**→**errorType()****watchdog.get_errorType()**

Returns the numerical error code of the latest error with the watchdog.

```
function get_errorType( ): YRETCODE
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the watchdog object

watchdog→get_functionDescriptor()

YWatchdog

watchdog→functionDescriptor()

watchdog.get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**(): YFUN_DESCR

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

watchdog→get_logicalName()**YWatchdog****watchdog→logicalName()****watchdog.get_logicalName()**

Returns the logical name of the watchdog.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the watchdog. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

watchdog→**get_maxTimeOnStateA()**

YWatchdog

watchdog→**maxTimeOnStateA()**

watchdog.get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
function get_maxTimeOnStateA( ): int64
```

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEA_INVALID.

watchdog→get_maxTimeOnStateB()**YWatchdog****watchdog→maxTimeOnStateB()****watchdog.get_maxTimeOnStateB()**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
function get_maxTimeOnStateB( ): int64
```

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns Y_MAXTIMEONSTATEB_INVALID.

watchdog→**get_module()**

YWatchdog

watchdog→**module()****watchdog.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

watchdog→get_output()**YWatchdog****watchdog→output()watchdog.get_output()**

Returns the output state of the watchdog, when used as a simple switch (single throw).

function **get_output()**: Integer

Returns :

either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns Y_OUTPUT_INVALID.

watchdog→get_pulseTimer()

YWatchdog

watchdog→pulseTimer()watchdog.get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

```
function get_pulseTimer( ): int64
```

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y_PULSETIMER_INVALID.

watchdog→get_running()**YWatchdog****watchdog→running()watchdog.get_running()**

Returns the watchdog running state.

```
function get_running( ): Integer
```

Returns :

either Y_RUNNING_OFF or Y_RUNNING_ON, according to the watchdog running state

On failure, throws an exception or returns Y_RUNNING_INVALID.

watchdog→get_state()

YWatchdog

watchdog→state()watchdog.get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

function **get_state**(): Integer

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

watchdog→get_stateAtPowerOn()**YWatchdog****watchdog→stateAtPowerOn()****watchdog.get_stateAtPowerOn()**

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```
function get_stateAtPowerOn( ): Integer
```

Returns :

a value among Y_STATEATPOWERON_UNCHANGED, Y_STATEATPOWERON_A and Y_STATEATPOWERON_B corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y_STATEATPOWERON_INVALID.

watchdog→get_triggerDelay()

YWatchdog

watchdog→triggerDelay()

watchdog.get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

```
function get_triggerDelay( ): int64
```

Returns :

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns Y_TRIGGERDELAY_INVALID.

watchdog→get_triggerDuration()**YWatchdog****watchdog→triggerDuration()****watchdog.get_triggerDuration()**

Returns the duration of resets caused by the watchdog, in milliseconds.

```
function get_triggerDuration( ): int64
```

Returns :

an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns Y_TRIGGERDURATION_INVALID.

watchdog→**get_userData()**

YWatchdog

watchdog→**userData()****watchdog.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( ): Tobject
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

watchdog→isOnline()watchdog.isOnline()**YWatchdog**

Checks if the watchdog is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

Returns :

`true` if the watchdog can be reached, and `false` otherwise

watchdog→load()**watchdog.load()****YWatchdog**

Preloads the watchdog cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

watchdog→nextWatchdog()
watchdog.nextWatchdog()

YWatchdog

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

```
function nextWatchdog( ): TYWatchdog
```

Returns :

a pointer to a `YWatchdog` object, corresponding to a watchdog currently online, or a `null` pointer if there are no more watchdog to enumerate.

watchdog→pulse()watchdog.pulse()

YWatchdog

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
function pulse( ms_duration: LongInt): integer
```

Parameters :

ms_duration pulse duration, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→registerValueCallback()
watchdog.registerValueCallback()

YWatchdog

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYWatchdogValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

watchdog→resetWatchdog()
watchdog.resetWatchdog()

YWatchdog

Resets the watchdog.

```
function resetWatchdog( ): integer
```

When the watchdog is running, this function must be called on a regular basis to prevent the watchdog to trigger

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_autoStart()**YWatchdog****watchdog→setAutoStart()watchdog.set_autoStart()**

Changes the watchdog runningstae at module power on.

```
function set_autoStart( newval: Integer): integer
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval either Y_AUTOSTART_OFF or Y_AUTOSTART_ON, according to the watchdog runningstae at module power on

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→**set_logicalName()****YWatchdog****watchdog**→**setLogicalName()****watchdog.set_logicalName()**

Changes the logical name of the watchdog.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the watchdog.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateA()**YWatchdog****watchdog→setMaxTimeOnStateA()****watchdog.set_maxTimeOnStateA()**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
function set_maxTimeOnStateA( newval: int64): integer
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateB()

YWatchdog

watchdog→setMaxTimeOnStateB()

watchdog.set_maxTimeOnStateB()

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
function set_maxTimeOnStateB( newval: int64): integer
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_output()**YWatchdog****watchdog→setOutput()watchdog.set_output()**

Changes the output state of the watchdog, when used as a simple switch (single throw).

```
function set_output( newval: Integer): integer
```

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_running()

YWatchdog

watchdog→setRunning()watchdog.set_running()

Changes the running state of the watchdog.

```
function set_running( newval: Integer): integer
```

Parameters :

newval either Y_RUNNING_OFF or Y_RUNNING_ON, according to the running state of the watchdog

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_state()**YWatchdog****watchdog→setState()watchdog.set_state()**

Changes the state of the watchdog (A for the idle position, B for the active position).

```
function set_state( newval: Integer): integer
```

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→**set_stateAtPowerOn()****YWatchdog****watchdog**→**setStateAtPowerOn()****watchdog.set_stateAtPowerOn()**

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
function set_stateAtPowerOn( newval: Integer): integer
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDelay()
watchdog→setTriggerDelay()
watchdog.set_triggerDelay()

YWatchdog

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

```
function set_triggerDelay( newval: int64): integer
```

Parameters :

newval an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDuration()**YWatchdog****watchdog→setTriggerDuration()****watchdog.set_triggerDuration()**

Changes the duration of resets caused by the watchdog, in milliseconds.

```
function set_triggerDuration( newval: int64): integer
```

Parameters :

newval an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_userdata()**YWatchdog****watchdog→setUserData()watchdog.set_userdata()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userdata**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

3.46. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wireless.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YWireless = yoctolib.YWireless;
php	require_once('yocto_wireless.php');
c++	#include "yocto_wireless.h"
m	#import "yocto_wireless.h"
pas	uses yocto_wireless;
vb	yocto_wireless.vb
cs	yocto_wireless.cs
java	import com.yoctopuce.YoctoAPI.YWireless;
py	from yocto_wireless import *

Global functions
yFindWireless(func) Retrieves a wireless lan interface for a given identifier.
yFirstWireless() Starts the enumeration of wireless lan interfaces currently accessible.
YWireless methods
wireless→adhocNetwork(ssid, securityKey) Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.
wireless→describe() Returns a short text that describes unambiguously the instance of the wireless lan interface in the form TYPE (NAME) = SERIAL . FUNCTIONID.
wireless→get_advertisedValue() Returns the current value of the wireless lan interface (no more than 6 characters).
wireless→get_channel() Returns the 802.11 channel currently used, or 0 when the selected network has not been found.
wireless→get_detectedWlans() Returns a list of YWlanRecord objects that describe detected Wireless networks.
wireless→get_errorMessage() Returns the error message of the latest error with the wireless lan interface.
wireless→get_errorType() Returns the numerical error code of the latest error with the wireless lan interface.
wireless→get_friendlyName() Returns a global identifier of the wireless lan interface in the format MODULE_NAME . FUNCTION_NAME.
wireless→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
wireless→get_functionId() Returns the hardware identifier of the wireless lan interface, without reference to the module.
wireless→get_hardwareId() Returns the unique hardware identifier of the wireless lan interface in the form SERIAL . FUNCTIONID.

wireless→get_linkQuality()
Returns the link quality, expressed in percent.
wireless→get_logicalName()
Returns the logical name of the wireless lan interface.
wireless→get_message()
Returns the latest status message from the wireless interface.
wireless→get_module()
Gets the YModule object for the device on which the function is located.
wireless→get_module_async(callback, context)
Gets the YModule object for the device on which the function is located (asynchronous version).
wireless→get_security()
Returns the security algorithm used by the selected wireless network.
wireless→get_ssid()
Returns the wireless network name (SSID).
wireless→get_userData()
Returns the value of the userData attribute, as previously stored using method set_userData.
wireless→isOnline()
Checks if the wireless lan interface is currently reachable, without raising any error.
wireless→isOnline_async(callback, context)
Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).
wireless→joinNetwork(ssid, securityKey)
Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).
wireless→load(msValidity)
Preloads the wireless lan interface cache with a specified validity duration.
wireless→load_async(msValidity, callback, context)
Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).
wireless→nextWireless()
Continues the enumeration of wireless lan interfaces started using yFirstWireless().
wireless→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.
wireless→set_logicalName(newval)
Changes the logical name of the wireless lan interface.
wireless→set_userData(data)
Stores a user context provided as argument in the userData attribute of the function.
wireless→wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWireless.FindWireless() yFindWireless()yFindWireless()

YWireless

Retrieves a wireless lan interface for a given identifier.

```
function yFindWireless( func: string): TYWireless
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wireless lan interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWireless.isOnline()` to test if the wireless lan interface is indeed online at a given time. In case of ambiguity when looking for a wireless lan interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the wireless lan interface

Returns :

a `YWireless` object allowing you to drive the wireless lan interface.

**YWireless.FirstWireless()
yFirstWireless()yFirstWireless()**

YWireless

Starts the enumeration of wireless lan interfaces currently accessible.

```
function yFirstWireless( ): TYWireless
```

Use the method `YWireless.nextWireless()` to iterate on next wireless lan interfaces.

Returns :

a pointer to a `YWireless` object, corresponding to the first wireless lan interface currently online, or a `null` pointer if there are none.

wireless→adhocNetwork()wireless.adhocNetwork()**YWireless**

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

```
function adhocNetwork( ssid: string, securityKey: string): integer
```

If a security key is specified, the network is protected by WEP128, since WPA is not standardized for ad-hoc networks. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid the name of the network to connect to
securityKey the network key, as a character string

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→describe()wireless.describe()**YWireless**

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function describe( ): string
```

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or "unresolved", and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the wireless lan interface (ex:
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wireless→**get_advertisedValue()**

YWireless

wireless→**advertisedValue()**

wireless.get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

```
function get_advertisedValue( ): string
```

Returns :

a string corresponding to the current value of the wireless lan interface (no more than 6 characters). On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

wireless→**get_channel()****YWireless****wireless**→**channel()****wireless.get_channel()**

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

function **get_channel()**: LongInt

Returns :

an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns Y_CHANNEL_INVALID.

wireless→get_detectedWlans()

YWireless

wireless→detectedWlans()

wireless.get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

```
function get_detectedWlans( ): TYWlanRecordArray
```

This list is not updated when the module is already connected to an access point (infrastructure mode). To force an update of this list, `adhocNetwork()` must be called to disconnect the module from the current network. The returned list must be unallocated by the caller.

Returns :

a list of YWlanRecord objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.

wireless→get_errorMessage()
wireless→errorMessage()
wireless.get_errorMessage()

YWireless

Returns the error message of the latest error with the wireless lan interface.

```
function get_errorMessage( ): string
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the wireless lan interface object

wireless→**get_errorType()**

YWireless

wireless→**errorType()****wireless.get_errorType()**

Returns the numerical error code of the latest error with the wireless lan interface.

function **get_errorType**(): YRETCODE

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the wireless lan interface object

wireless→get_functionDescriptor()
wireless→functionDescriptor()
wireless.get_functionDescriptor()

YWireless

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

```
function get_functionDescriptor( ): YFUN_DESCR
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

wireless→get_linkQuality()

YWireless

wireless→linkQuality()wireless.get_linkQuality()

Returns the link quality, expressed in percent.

```
function get_linkQuality( ): LongInt
```

Returns :

an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns Y_LINKQUALITY_INVALID.

wireless→**get_logicalName()****YWireless****wireless**→**logicalName()****wireless.get_logicalName()**

Returns the logical name of the wireless lan interface.

```
function get_logicalName( ): string
```

Returns :

a string corresponding to the logical name of the wireless lan interface. On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

wireless→get_message()

YWireless

wireless→message()wireless.get_message()

Returns the latest status message from the wireless interface.

```
function get_message( ): string
```

Returns :

a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns Y_MESSAGE_INVALID.

wireless→**get_module()****YWireless****wireless**→**module()****wireless.get_module()**

Gets the YModule object for the device on which the function is located.

```
function get_module( ): TModule
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

wireless→get_security()

YWireless

wireless→security()wireless.get_security()

Returns the security algorithm used by the selected wireless network.

function **get_security**(): Integer

Returns :

a value among Y_SECURITY_UNKNOWN, Y_SECURITY_OPEN, Y_SECURITY_WEP, Y_SECURITY_WPA and Y_SECURITY_WPA2 corresponding to the security algorithm used by the selected wireless network

On failure, throws an exception or returns Y_SECURITY_INVALID.

wireless→**get_ssid()****YWireless****wireless**→**ssid()****wireless.get_ssid()**

Returns the wireless network name (SSID).

```
function get_ssid( ): string
```

Returns :

a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns Y_SSID_INVALID.

wireless→**get_userData()**

YWireless

wireless→**userData()****wireless.get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**(): Tobject

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

wireless→isOnline()wireless.isOnline()**YWireless**

Checks if the wireless lan interface is currently reachable, without raising any error.

```
function isOnline( ): boolean
```

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

Returns :

`true` if the wireless lan interface can be reached, and `false` otherwise

wireless→joinNetwork()wireless.joinNetwork()**YWireless**

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

```
function joinNetwork( ssid: string, securityKey: string): integer
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid the name of the network to connect to
securityKey the network key, as a character string

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→load()wireless.load()**YWireless**

Preloads the wireless lan interface cache with a specified validity duration.

```
function load( msValidity: integer): YRETCODE
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

wireless→**nextWireless()****wireless.nextWireless()**

YWireless

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

```
function nextWireless( ): TYWireless
```

Returns :

a pointer to a `YWireless` object, corresponding to a wireless lan interface currently online, or a `null` pointer if there are no more wireless lan interfaces to enumerate.

wireless→registerValueCallback()
wireless.registerValueCallback()

YWireless

Registers the callback function that is invoked on every change of advertised value.

```
function registerValueCallback( callback: TYWirelessValueCallback): LongInt
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wireless→**set_logicalName()****YWireless****wireless**→**setLogicalName()****wireless.set_logicalName()**

Changes the logical name of the wireless lan interface.

```
function set_logicalName( newval: string): integer
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wireless lan interface.

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wireless→**set_userData()****YWireless****wireless**→**setUserData()****wireless.set_userData()**

Stores a user context provided as argument in the userData attribute of the function.

procedure **set_userData**(**data**: Tobject)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

Index

A

Accelerometer 31
adhocNetwork, YWireless 1441
AnButton 70

B

Blueprint 10

C

calibrate, YLightSensor 637
calibrateFromPoints, YAccelerometer 35
calibrateFromPoints, YCarbonDioxide 109
calibrateFromPoints, YCompass 171
calibrateFromPoints, YCurrent 208
calibrateFromPoints, YGenericSensor 464
calibrateFromPoints, YGyro 507
calibrateFromPoints, YHumidity 577
calibrateFromPoints, YLightSensor 638
calibrateFromPoints, YMagnetometer 674
calibrateFromPoints, YPower 834
calibrateFromPoints, YPressure 874
calibrateFromPoints, YQt 966
calibrateFromPoints, YSensor 1091
calibrateFromPoints, YTemperature 1159
calibrateFromPoints, YTilt 1197
calibrateFromPoints, YVoc 1233
calibrateFromPoints, YVoltage 1269
callbackLogin, YNetwork 761
cancel3DCalibration, YRefFrame 1025
CarbonDioxide 105
CheckLogicalName, YAPI 12
clear, YDisplayLayer 382
clearConsole, YDisplayLayer 383
Clock 997
ColorLed 141
Compass 167
Configuration 1021
consoleOut, YDisplayLayer 384
copyLayerContent, YDisplay 342
Current 204

D

Data 268, 270, 282
DataLogger 240
delayedPulse, YDigitalIO 301
delayedPulse, YRelay 1058
delayedPulse, YWatchdog 1400
Delphi 3
describe, YAccelerometer 36
describe, YAnButton 74
describe, YCarbonDioxide 110
describe, YColorLed 144

describe, YCompass 172
describe, YCurrent 209
describe, YDataLogger 243
describe, YDigitalIO 302
describe, YDisplay 343
describe, YDualPower 416
describe, YFiles 438
describe, YGenericSensor 465
describe, YGyro 508
describe, YHubPort 554
describe, YHumidity 578
describe, YLed 612
describe, YLightSensor 639
describe, YMagnetometer 675
describe, YModule 719
describe, YNetwork 762
describe, YOsControl 813
describe, YPower 835
describe, YPressure 875
describe, YPwmOutput 910
describe, YPwmPowerSource 944
describe, YQt 966
describe, YRealTimeClock 1000
describe, YRefFrame 1026
describe, YRelay 1059
describe, YSensor 1092
describe, YServo 1127
describe, YTemperature 1160
describe, YTilt 1198
describe, YVoc 1234
describe, YVoltage 1270
describe, YVSource 1304
describe, YWakeUpMonitor 1334
describe, YWakeUpSchedule 1366
describe, YWatchdog 1401
describe, YWireless 1442
Digital 297
DisableExceptions, YAPI 13
Display 338
DisplayLayer 381
download, YFiles 439
download, YModule 720
drawBar, YDisplayLayer 385
drawBitmap, YDisplayLayer 386
drawCircle, YDisplayLayer 387
drawDisc, YDisplayLayer 388
drawImage, YDisplayLayer 389
drawPixel, YDisplayLayer 390
drawRect, YDisplayLayer 391
drawText, YDisplayLayer 392
dutyCycleMove, YPwmOutput 911

E

EnableExceptions, YAPI 14

Error 7
External 413

F

fade, YDisplay 344
Files 435
FindAccelerometer, YAccelerometer 33
FindAnButton, YAnButton 72
FindCarbonDioxide, YCarbonDioxide 107
FindColorLed, YColorLed 142
FindCompass, YCompass 169
FindCurrent, YCurrent 206
FindDataLogger, YDataLogger 241
FindDigitalIO, YDigitalIO 299
FindDisplay, YDisplay 340
FindDualPower, YDualPower 414
FindFiles, YFiles 436
FindGenericSensor, YGenericSensor 462
FindGyro, YGyro 505
FindHubPort, YHubPort 552
FindHumidity, YHumidity 575
FindLed, YLed 610
FindLightSensor, YLightSensor 635
FindMagnetometer, YMagnetometer 672
FindModule, YModule 717
FindNetwork, YNetwork 759
FindOsControl, YOsControl 811
FindPower, YPower 832
FindPressure, YPressure 872
FindPwmOutput, YPwmOutput 908
FindPwmPowerSource, YPwmPowerSource 942
FindQt, YQt 963
FindRealTimeClock, YRealTimeClock 998
FindRefFrame, YRefFrame 1023
FindRelay, YRelay 1056
FindSensor, YSensor 1089
FindServo, YServo 1125
FindTemperature, YTemperature 1157
FindTilt, YTilt 1195
FindVoc, YVoc 1231
FindVoltage, YVoltage 1267
FindVSource, YVSource 1302
FindWakeUpMonitor, YWakeUpMonitor 1332
FindWakeUpSchedule, YWakeUpSchedule 1364
FindWatchdog, YWatchdog 1398
FindWireless, YWireless 1439
FirstAccelerometer, YAccelerometer 34
FirstAnButton, YAnButton 73
FirstCarbonDioxide, YCarbonDioxide 108
FirstColorLed, YColorLed 143
FirstCompass, YCompass 170
FirstCurrent, YCurrent 207
FirstDataLogger, YDataLogger 242
FirstDigitalIO, YDigitalIO 300
FirstDisplay, YDisplay 341
FirstDualPower, YDualPower 415
FirstFiles, YFiles 437
FirstGenericSensor, YGenericSensor 463
FirstGyro, YGyro 506

FirstHubPort, YHubPort 553
FirstHumidity, YHumidity 576
FirstLed, YLed 611
FirstLightSensor, YLightSensor 636
FirstMagnetometer, YMagnetometer 673
FirstModule, YModule 718
FirstNetwork, YNetwork 760
FirstOsControl, YOsControl 812
FirstPower, YPower 833
FirstPressure, YPressure 873
FirstPwmOutput, YPwmOutput 909
FirstPwmPowerSource, YPwmPowerSource 943
FirstQt, YQt 964
FirstRealTimeClock, YRealTimeClock 999
FirstRefFrame, YRefFrame 1024
FirstRelay, YRelay 1057
FirstSensor, YSensor 1090
FirstServo, YServo 1126
FirstTemperature, YTemperature 1158
FirstTilt, YTilt 1196
FirstVoc, YVoc 1232
FirstVoltage, YVoltage 1268
FirstVSource, YVSource 1303
FirstWakeUpMonitor, YWakeUpMonitor 1333
FirstWakeUpSchedule, YWakeUpSchedule 1365
FirstWatchdog, YWatchdog 1399
FirstWireless, YWireless 1440
forgetAllDataStreams, YDataLogger 244
format_fs, YFiles 440
Formatted 268
Frame 1021
FreeAPI, YAPI 15
functionCount, YModule 721
functionId, YModule 722
functionName, YModule 723
Functions 11
functionValue, YModule 724

G

General 11
GenericSensor 460
get_3DCalibrationHint, YRefFrame 1027
get_3DCalibrationLogMsg, YRefFrame 1028
get_3DCalibrationProgress, YRefFrame 1029
get_3DCalibrationStage, YRefFrame 1030
get_3DCalibrationStageProgress, YRefFrame 1031
get_adminPassword, YNetwork 763
get_advertisedValue, YAccelerometer 37
get_advertisedValue, YAnButton 75
get_advertisedValue, YCarbonDioxide 111
get_advertisedValue, YColorLed 145
get_advertisedValue, YCompass 173
get_advertisedValue, YCurrent 210
get_advertisedValue, YDataLogger 245
get_advertisedValue, YDigitalIO 303
get_advertisedValue, YDisplay 345
get_advertisedValue, YDualPower 417
get_advertisedValue, YFiles 441

get_advertisedValue, YGenericSensor 466
 get_advertisedValue, YGyro 509
 get_advertisedValue, YHubPort 555
 get_advertisedValue, YHumidity 579
 get_advertisedValue, YLed 613
 get_advertisedValue, YLightSensor 640
 get_advertisedValue, YMagnetometer 676
 get_advertisedValue, YNetwork 764
 get_advertisedValue, YOsControl 814
 get_advertisedValue, YPower 836
 get_advertisedValue, YPressure 876
 get_advertisedValue, YPwmOutput 912
 get_advertisedValue, YPwmPowerSource 945
 get_advertisedValue, YQt 967
 get_advertisedValue, YRealTimeClock 1001
 get_advertisedValue, YRefFrame 1032
 get_advertisedValue, YRelay 1060
 get_advertisedValue, YSensor 1093
 get_advertisedValue, YServo 1128
 get_advertisedValue, YTemperature 1161
 get_advertisedValue, YTilt 1199
 get_advertisedValue, YVoc 1235
 get_advertisedValue, YVoltage 1271
 get_advertisedValue, YVSource 1305
 get_advertisedValue, YWakeUpMonitor 1335
 get_advertisedValue, YWakeUpSchedule 1367
 get_advertisedValue, YWatchdog 1402
 get_advertisedValue, YWireless 1443
 get_analogCalibration, YAnButton 76
 get_autoStart, YDataLogger 246
 get_autoStart, YWatchdog 1403
 get_averageValue, YDataStream 283
 get_averageValue, YMeasure 709
 get_baudRate, YHubPort 556
 get_beacon, YModule 725
 get_bearing, YRefFrame 1033
 get_bitDirection, YDigitalIO 304
 get_bitOpenDrain, YDigitalIO 305
 get_bitPolarity, YDigitalIO 306
 get_bitState, YDigitalIO 307
 get_blinking, YLed 614
 get_brightness, YDisplay 346
 get_calibratedValue, YAnButton 77
 get_calibrationMax, YAnButton 78
 get_calibrationMin, YAnButton 79
 get_callbackCredentials, YNetwork 765
 get_callbackEncoding, YNetwork 766
 get_callbackMaxDelay, YNetwork 767
 get_callbackMethod, YNetwork 768
 get_callbackMinDelay, YNetwork 769
 get_callbackUrl, YNetwork 770
 get_channel, YWireless 1444
 get_columnCount, YDataStream 284
 get_columnNames, YDataStream 285
 get_cosPhi, YPower 837
 get_countdown, YRelay 1061
 get_countdown, YWatchdog 1404
 get_currentRawValue, YAccelerometer 38
 get_currentRawValue, YCarbonDioxide 112
 get_currentRawValue, YCompass 174
 get_currentRawValue, YCurrent 211
 get_currentRawValue, YGenericSensor 467
 get_currentRawValue, YGyro 510
 get_currentRawValue, YHumidity 580
 get_currentRawValue, YLightSensor 641
 get_currentRawValue, YMagnetometer 677
 get_currentRawValue, YPower 838
 get_currentRawValue, YPressure 877
 get_currentRawValue, YQt 968
 get_currentRawValue, YSensor 1094
 get_currentRawValue, YTemperature 1162
 get_currentRawValue, YTilt 1200
 get_currentRawValue, YVoc 1236
 get_currentRawValue, YVoltage 1272
 get_currentRunIndex, YDataLogger 247
 get_currentValue, YAccelerometer 39
 get_currentValue, YCarbonDioxide 113
 get_currentValue, YCompass 175
 get_currentValue, YCurrent 212
 get_currentValue, YGenericSensor 468
 get_currentValue, YGyro 511
 get_currentValue, YHumidity 581
 get_currentValue, YLightSensor 642
 get_currentValue, YMagnetometer 678
 get_currentValue, YPower 839
 get_currentValue, YPressure 878
 get_currentValue, YQt 969
 get_currentValue, YSensor 1095
 get_currentValue, YTemperature 1163
 get_currentValue, YTilt 1201
 get_currentValue, YVoc 1237
 get_currentValue, YVoltage 1273
 get_data, YDataStream 286
 get_dataRows, YDataStream 287
 get_dataSamplesIntervalMs, YDataStream 288
 get_dataSets, YDataLogger 248
 get_dataStreams, YDataLogger 249
 get_dateTime, YRealTimeClock 1002
 get_detectedWlans, YWireless 1445
 get_discoverable, YNetwork 771
 get_display, YDisplayLayer 393
 get_displayHeight, YDisplay 347
 get_displayHeight, YDisplayLayer 394
 get_displayType, YDisplay 348
 get_displayWidth, YDisplay 349
 get_displayWidth, YDisplayLayer 395
 get_duration, YDataStream 289
 get_dutyCycle, YPwmOutput 913
 get_dutyCycleAtPowerOn, YPwmOutput 914
 get_enabled, YDisplay 350
 get_enabled, YHubPort 557
 get_enabled, YPwmOutput 915
 get_enabled, YServo 1129
 get_enabledAtPowerOn, YPwmOutput 916
 get_enabledAtPowerOn, YServo 1130
 get_endTimeUTC, YDataSet 271
 get_endTimeUTC, YMeasure 710
 get_errorMessage, YAccelerometer 40

get_errorMessage, YAnButton 80
get_errorMessage, YCarbonDioxide 114
get_errorMessage, YColorLed 146
get_errorMessage, YCompass 176
get_errorMessage, YCurrent 213
get_errorMessage, YDataLogger 250
get_errorMessage, YDigitalIO 308
get_errorMessage, YDisplay 351
get_errorMessage, YDualPower 418
get_errorMessage, YFiles 442
get_errorMessage, YGenericSensor 469
get_errorMessage, YGyro 512
get_errorMessage, YHubPort 558
get_errorMessage, YHumidity 582
get_errorMessage, YLed 615
get_errorMessage, YLightSensor 643
get_errorMessage, YMagnetometer 679
get_errorMessage, YModule 726
get_errorMessage, YNetwork 772
get_errorMessage, YOsControl 815
get_errorMessage, YPower 840
get_errorMessage, YPressure 879
get_errorMessage, YPwmOutput 917
get_errorMessage, YPwmPowerSource 946
get_errorMessage, YQt 970
get_errorMessage, YRealTimeClock 1003
get_errorMessage, YRefFrame 1034
get_errorMessage, YRelay 1062
get_errorMessage, YSensor 1096
get_errorMessage, YServo 1131
get_errorMessage, YTemperature 1164
get_errorMessage, YTilt 1202
get_errorMessage, YVoc 1238
get_errorMessage, YVoltage 1274
get_errorMessage, YVSource 1306
get_errorMessage, YWakeUpMonitor 1336
get_errorMessage, YWakeUpSchedule 1368
get_errorMessage, YWatchdog 1405
get_errorMessage, YWireless 1446
get_errorType, YAccelerometer 41
get_errorType, YAnButton 81
get_errorType, YCarbonDioxide 115
get_errorType, YColorLed 147
get_errorType, YCompass 177
get_errorType, YCurrent 214
get_errorType, YDataLogger 251
get_errorType, YDigitalIO 309
get_errorType, YDisplay 352
get_errorType, YDualPower 419
get_errorType, YFiles 443
get_errorType, YGenericSensor 470
get_errorType, YGyro 513
get_errorType, YHubPort 559
get_errorType, YHumidity 583
get_errorType, YLed 616
get_errorType, YLightSensor 644
get_errorType, YMagnetometer 680
get_errorType, YModule 727
get_errorType, YNetwork 773

get_errorType, YOsControl 816
get_errorType, YPower 841
get_errorType, YPressure 880
get_errorType, YPwmOutput 918
get_errorType, YPwmPowerSource 947
get_errorType, YQt 971
get_errorType, YRealTimeClock 1004
get_errorType, YRefFrame 1035
get_errorType, YRelay 1063
get_errorType, YSensor 1097
get_errorType, YServo 1132
get_errorType, YTemperature 1165
get_errorType, YTilt 1203
get_errorType, YVoc 1239
get_errorType, YVoltage 1275
get_errorType, YVSource 1307
get_errorType, YWakeUpMonitor 1337
get_errorType, YWakeUpSchedule 1369
get_errorType, YWatchdog 1406
get_errorType, YWireless 1447
get_extPowerFailure, YVSource 1308
get_extVoltage, YDualPower 420
get_failure, YVSource 1309
get_filesCount, YFiles 444
get_firmwareRelease, YModule 728
get_freeSpace, YFiles 445
get_frequency, YPwmOutput 919
get_functionDescriptor, YAccelerometer 42
get_functionDescriptor, YAnButton 82
get_functionDescriptor, YCarbonDioxide 116
get_functionDescriptor, YColorLed 148
get_functionDescriptor, YCompass 178
get_functionDescriptor, YCurrent 215
get_functionDescriptor, YDataLogger 252
get_functionDescriptor, YDigitalIO 310
get_functionDescriptor, YDisplay 353
get_functionDescriptor, YDualPower 421
get_functionDescriptor, YFiles 446
get_functionDescriptor, YGenericSensor 471
get_functionDescriptor, YGyro 514
get_functionDescriptor, YHubPort 560
get_functionDescriptor, YHumidity 584
get_functionDescriptor, YLed 617
get_functionDescriptor, YLightSensor 645
get_functionDescriptor, YMagnetometer 681
get_functionDescriptor, YNetwork 774
get_functionDescriptor, YOsControl 817
get_functionDescriptor, YPower 842
get_functionDescriptor, YPressure 881
get_functionDescriptor, YPwmOutput 920
get_functionDescriptor, YPwmPowerSource 948
get_functionDescriptor, YQt 972
get_functionDescriptor, YRealTimeClock 1005
get_functionDescriptor, YRefFrame 1036
get_functionDescriptor, YRelay 1064
get_functionDescriptor, YSensor 1098
get_functionDescriptor, YServo 1133
get_functionDescriptor, YTemperature 1166
get_functionDescriptor, YTilt 1204

get_functionDescriptor, YVoc 1240
get_functionDescriptor, YVoltage 1276
get_functionDescriptor, YVSource 1310
get_functionDescriptor, YWakeUpMonitor 1338
get_functionDescriptor, YWakeUpSchedule 1370
get_functionDescriptor, YWatchdog 1407
get_functionDescriptor, YWireless 1448
get_functionId, YDataSet 272
get_hardwareId, YDataSet 273
get_heading, YGyro 515
get_highestValue, YAccelerometer 43
get_highestValue, YCarbonDioxide 117
get_highestValue, YCompass 179
get_highestValue, YCurrent 216
get_highestValue, YGenericSensor 472
get_highestValue, YGyro 516
get_highestValue, YHumidity 585
get_highestValue, YLightSensor 646
get_highestValue, YMagnetometer 682
get_highestValue, YPower 843
get_highestValue, YPressure 882
get_highestValue, YQt 973
get_highestValue, YSensor 1099
get_highestValue, YTemperature 1167
get_highestValue, YTilt 1205
get_highestValue, YVoc 1241
get_highestValue, YVoltage 1277
get_hours, YWakeUpSchedule 1371
get_hslColor, YColorLed 149
get_icon2d, YModule 729
get_ipAddress, YNetwork 775
get_isPressed, YAnButton 83
get_lastLogs, YModule 730
get_lastTimePressed, YAnButton 84
get_lastTimeReleased, YAnButton 85
get_layerCount, YDisplay 354
get_layerHeight, YDisplay 355
get_layerHeight, YDisplayLayer 396
get_layerWidth, YDisplay 356
get_layerWidth, YDisplayLayer 397
get_linkQuality, YWireless 1449
get_list, YFiles 447
get_logFrequency, YAccelerometer 44
get_logFrequency, YCarbonDioxide 118
get_logFrequency, YCompass 180
get_logFrequency, YCurrent 217
get_logFrequency, YGenericSensor 473
get_logFrequency, YGyro 517
get_logFrequency, YHumidity 586
get_logFrequency, YLightSensor 647
get_logFrequency, YMagnetometer 683
get_logFrequency, YPower 844
get_logFrequency, YPressure 883
get_logFrequency, YQt 974
get_logFrequency, YSensor 1100
get_logFrequency, YTemperature 1168
get_logFrequency, YTilt 1206
get_logFrequency, YVoc 1242
get_logFrequency, YVoltage 1278

get_logicalName, YAccelerometer 45
get_logicalName, YAnButton 86
get_logicalName, YCarbonDioxide 119
get_logicalName, YColorLed 150
get_logicalName, YCompass 181
get_logicalName, YCurrent 218
get_logicalName, YDataLogger 253
get_logicalName, YDigitalIO 311
get_logicalName, YDisplay 357
get_logicalName, YDualPower 422
get_logicalName, YFiles 448
get_logicalName, YGenericSensor 474
get_logicalName, YGyro 518
get_logicalName, YHubPort 561
get_logicalName, YHumidity 587
get_logicalName, YLed 618
get_logicalName, YLightSensor 648
get_logicalName, YMagnetometer 684
get_logicalName, YModule 731
get_logicalName, YNetwork 776
get_logicalName, YOsControl 818
get_logicalName, YPower 845
get_logicalName, YPressure 884
get_logicalName, YPwmOutput 921
get_logicalName, YPwmPowerSource 949
get_logicalName, YQt 975
get_logicalName, YRealTimeClock 1006
get_logicalName, YRefFrame 1037
get_logicalName, YRelay 1065
get_logicalName, YSensor 1101
get_logicalName, YServo 1134
get_logicalName, YTemperature 1169
get_logicalName, YTilt 1207
get_logicalName, YVoc 1243
get_logicalName, YVoltage 1279
get_logicalName, YVSource 1311
get_logicalName, YWakeUpMonitor 1339
get_logicalName, YWakeUpSchedule 1372
get_logicalName, YWatchdog 1408
get_logicalName, YWireless 1450
get_lowestValue, YAccelerometer 46
get_lowestValue, YCarbonDioxide 120
get_lowestValue, YCompass 182
get_lowestValue, YCurrent 219
get_lowestValue, YGenericSensor 475
get_lowestValue, YGyro 519
get_lowestValue, YHumidity 588
get_lowestValue, YLightSensor 649
get_lowestValue, YMagnetometer 685
get_lowestValue, YPower 846
get_lowestValue, YPressure 885
get_lowestValue, YQt 976
get_lowestValue, YSensor 1102
get_lowestValue, YTemperature 1170
get_lowestValue, YTilt 1208
get_lowestValue, YVoc 1244
get_lowestValue, YVoltage 1280
get_luminosity, YLed 619
get_luminosity, YModule 732

get_macAddress, YNetwork 777
get_magneticHeading, YCompass 183
get_maxTimeOnStateA, YRelay 1066
get_maxTimeOnStateA, YWatchdog 1409
get_maxTimeOnStateB, YRelay 1067
get_maxTimeOnStateB, YWatchdog 1410
get_maxValue, YDataStream 290
get_maxValue, YMeasure 711
get_measures, YDataSet 274
get_message, YWireless 1451
get_meter, YPower 847
get_meterTimer, YPower 848
get_minutes, YWakeUpSchedule 1373
get_minutesA, YWakeUpSchedule 1374
get_minutesB, YWakeUpSchedule 1375
get_minValue, YDataStream 291
get_minValue, YMeasure 712
get_module, YAccelerometer 47
get_module, YAnButton 87
get_module, YCarbonDioxide 121
get_module, YColorLed 151
get_module, YCompass 184
get_module, YCurrent 220
get_module, YDataLogger 254
get_module, YDigitalIO 312
get_module, YDisplay 358
get_module, YDualPower 423
get_module, YFiles 449
get_module, YGenericSensor 476
get_module, YGyro 520
get_module, YHubPort 562
get_module, YHumidity 589
get_module, YLed 620
get_module, YLightSensor 650
get_module, YMagnetometer 686
get_module, YNetwork 778
get_module, YOsControl 819
get_module, YPower 849
get_module, YPressure 886
get_module, YPwmOutput 922
get_module, YPwmPowerSource 950
get_module, YQt 977
get_module, YRealTimeClock 1007
get_module, YRefFrame 1038
get_module, YRelay 1068
get_module, YSensor 1103
get_module, YServo 1135
get_module, YTemperature 1171
get_module, YTilt 1209
get_module, YVoc 1245
get_module, YVoltage 1281
get_module, YVSource 1312
get_module, YWakeUpMonitor 1340
get_module, YWakeUpSchedule 1376
get_module, YWatchdog 1411
get_module, YWireless 1452
get_monthDays, YWakeUpSchedule 1377
get_months, YWakeUpSchedule 1378
get_mountOrientation, YRefFrame 1039

get_mountPosition, YRefFrame 1040
get_neutral, YServo 1136
get_nextOccurrence, YWakeUpSchedule 1379
get_nextWakeUp, YWakeUpMonitor 1341
get_orientation, YDisplay 359
get_output, YRelay 1069
get_output, YWatchdog 1412
get_outputVoltage, YDigitalIO 313
get_overCurrent, YVSource 1313
get_overHeat, YVSource 1314
get_overLoad, YVSource 1315
get_period, YPwmOutput 923
get_persistentSettings, YModule 733
get_pitch, YGyro 521
get_poeCurrent, YNetwork 779
get_portDirection, YDigitalIO 314
get_portOpenDrain, YDigitalIO 315
get_portPolarity, YDigitalIO 316
get_portSize, YDigitalIO 317
get_portState, YDigitalIO 318
get_portState, YHubPort 563
get_position, YServo 1137
get_positionAtPowerOn, YServo 1138
get_power, YLed 621
get_powerControl, YDualPower 424
get_powerDuration, YWakeUpMonitor 1342
get_powerMode, YPwmPowerSource 951
get_powerState, YDualPower 425
get_preview, YDataSet 275
get_primaryDNS, YNetwork 780
get_productId, YModule 734
get_productName, YModule 735
get_productRelease, YModule 736
get_progress, YDataSet 276
get_pulseCounter, YAnButton 88
get_pulseDuration, YPwmOutput 924
get_pulseTimer, YAnButton 89
get_pulseTimer, YRelay 1070
get_pulseTimer, YWatchdog 1413
get_quaternionW, YGyro 522
get_quaternionX, YGyro 523
get_quaternionY, YGyro 524
get_quaternionZ, YGyro 525
get_range, YServo 1139
get_rawValue, YAnButton 90
get_readiness, YNetwork 781
get_rebootCountdown, YModule 737
get_recordedData, YAccelerometer 48
get_recordedData, YCarbonDioxide 122
get_recordedData, YCompass 185
get_recordedData, YCurrent 221
get_recordedData, YGenericSensor 477
get_recordedData, YGyro 526
get_recordedData, YHumidity 590
get_recordedData, YLightSensor 651
get_recordedData, YMagnetometer 687
get_recordedData, YPower 850
get_recordedData, YPressure 887
get_recordedData, YQt 978

get_recordedData, YSensor 1104
get_recordedData, YTemperature 1172
get_recordedData, YTilt 1210
get_recordedData, YVoc 1246
get_recordedData, YVoltage 1282
get_recording, YDataLogger 255
get_regulationFailure, YVSource 1316
get_reportFrequency, YAccelerometer 49
get_reportFrequency, YCarbonDioxide 123
get_reportFrequency, YCompass 186
get_reportFrequency, YCurrent 222
get_reportFrequency, YGenericSensor 478
get_reportFrequency, YGyro 527
get_reportFrequency, YHumidity 591
get_reportFrequency, YLightSensor 652
get_reportFrequency, YMagnetometer 688
get_reportFrequency, YPower 851
get_reportFrequency, YPressure 888
get_reportFrequency, YQt 979
get_reportFrequency, YSensor 1105
get_reportFrequency, YTemperature 1173
get_reportFrequency, YTilt 1211
get_reportFrequency, YVoc 1247
get_reportFrequency, YVoltage 1283
get_resolution, YAccelerometer 50
get_resolution, YCarbonDioxide 124
get_resolution, YCompass 187
get_resolution, YCurrent 223
get_resolution, YGenericSensor 479
get_resolution, YGyro 528
get_resolution, YHumidity 592
get_resolution, YLightSensor 653
get_resolution, YMagnetometer 689
get_resolution, YPower 852
get_resolution, YPressure 889
get_resolution, YQt 980
get_resolution, YSensor 1106
get_resolution, YTemperature 1174
get_resolution, YTilt 1212
get_resolution, YVoc 1248
get_resolution, YVoltage 1284
get_rgbColor, YColorLed 152
get_rgbColorAtPowerOn, YColorLed 153
get_roll, YGyro 529
get_router, YNetwork 782
get_rowCount, YDataStream 292
get_runIndex, YDataStream 293
get_running, YWatchdog 1414
get_secondaryDNS, YNetwork 783
get_security, YWireless 1453
get_sensitivity, YAnButton 91
get_sensorType, YTemperature 1175
get_serialNumber, YModule 738
get_shutdownCountdown, YOsControl 820
get_signalRange, YGenericSensor 480
get_signalUnit, YGenericSensor 481
get_signalValue, YGenericSensor 482
get_sleepCountdown, YWakeUpMonitor 1343
get_ssid, YWireless 1454

get_startTime, YDataStream 294
get_startTimeUTC, YDataRun 268
get_startTimeUTC, YDataSet 277
get_startTimeUTC, YDataStream 295
get_startTimeUTC, YMeasure 713
get_startupSeq, YDisplay 360
get_state, YRelay 1071
get_state, YWatchdog 1415
get_stateAtPowerOn, YRelay 1072
get_stateAtPowerOn, YWatchdog 1416
get_subnetMask, YNetwork 784
get_summary, YDataSet 278
get_timeSet, YRealTimeClock 1008
get_timeUTC, YDataLogger 256
get_triggerDelay, YWatchdog 1417
get_triggerDuration, YWatchdog 1418
get_unit, YAccelerometer 51
get_unit, YCarbonDioxide 125
get_unit, YCompass 188
get_unit, YCurrent 224
get_unit, YDataSet 279
get_unit, YGenericSensor 483
get_unit, YGyro 530
get_unit, YHumidity 593
get_unit, YLightSensor 654
get_unit, YMagnetometer 690
get_unit, YPower 853
get_unit, YPressure 890
get_unit, YQt 981
get_unit, YSensor 1107
get_unit, YTemperature 1176
get_unit, YTilt 1213
get_unit, YVoc 1249
get_unit, YVoltage 1285
get_unit, YVSource 1317
get_unixTime, YRealTimeClock 1009
get_upTime, YModule 739
get_usbBandwidth, YModule 740
get_usbCurrent, YModule 741
get_userData, YAccelerometer 52
get_userData, YAnButton 92
get_userData, YCarbonDioxide 126
get_userData, YColorLed 154
get_userData, YCompass 189
get_userData, YCurrent 225
get_userData, YDataLogger 257
get_userData, YDigitalIO 319
get_userData, YDisplay 361
get_userData, YDualPower 426
get_userData, YFiles 450
get_userData, YGenericSensor 484
get_userData, YGyro 531
get_userData, YHubPort 564
get_userData, YHumidity 594
get_userData, YLed 622
get_userData, YLightSensor 655
get_userData, YMagnetometer 691
get_userData, YModule 742
get_userData, YNetwork 785

- get_userdata, YOsControl 821
- get_userdata, YPower 854
- get_userdata, YPressure 891
- get_userdata, YPwmOutput 925
- get_userdata, YPwmPowerSource 952
- get_userdata, YQt 982
- get_userdata, YRealTimeClock 1010
- get_userdata, YRefFrame 1041
- get_userdata, YRelay 1073
- get_userdata, YSensor 1108
- get_userdata, YServo 1140
- get_userdata, YTemperature 1177
- get_userdata, YTilt 1214
- get_userdata, YVoc 1250
- get_userdata, YVoltage 1286
- get_userdata, YVSource 1318
- get_userdata, YWakeUpMonitor 1344
- get_userdata, YWakeUpSchedule 1380
- get_userdata, YWatchdog 1419
- get_userdata, YWireless 1455
- get_userPassword, YNetwork 786
- get_utcOffset, YRealTimeClock 1011
- get_valueRange, YGenericSensor 485
- get_voltage, YVSource 1319
- get_wakeUpReason, YWakeUpMonitor 1345
- get_wakeUpState, YWakeUpMonitor 1346
- get_weekDays, YWakeUpSchedule 1381
- get_wwwWatchdogDelay, YNetwork 787
- get_xValue, YAccelerometer 53
- get_xValue, YGyro 532
- get_xValue, YMagnetometer 692
- get_yValue, YAccelerometer 54
- get_yValue, YGyro 533
- get_yValue, YMagnetometer 693
- get_zValue, YAccelerometer 55
- get_zValue, YGyro 534
- get_zValue, YMagnetometer 694
- GetAPIVersion, YAPI 16
- GetTickCount, YAPI 17
- Gyroscope 503

H

- HandleEvents, YAPI 18
- hide, YDisplayLayer 398
- hslMove, YColorLed 155
- Humidity 573

I

- InitAPI, YAPI 19
- Interface 31, 70, 105, 141, 167, 204, 240, 297, 338, 381, 413, 435, 460, 503, 551, 573, 609, 633, 670, 715, 756, 830, 870, 906, 941, 961, 997, 1054, 1087, 1123, 1155, 1193, 1229, 1265, 1301, 1330, 1362, 1396, 1438
- Introduction 1
- isOnline, YAccelerometer 56
- isOnline, YAnButton 93
- isOnline, YCarbonDioxide 127

- isOnline, YColorLed 156
- isOnline, YCompass 190
- isOnline, YCurrent 226
- isOnline, YDataLogger 258
- isOnline, YDigitalIO 320
- isOnline, YDisplay 362
- isOnline, YDualPower 427
- isOnline, YFiles 451
- isOnline, YGenericSensor 486
- isOnline, YGyro 535
- isOnline, YHubPort 565
- isOnline, YHumidity 595
- isOnline, YLed 623
- isOnline, YLightSensor 656
- isOnline, YMagnetometer 695
- isOnline, YModule 743
- isOnline, YNetwork 788
- isOnline, YOsControl 822
- isOnline, YPower 855
- isOnline, YPressure 892
- isOnline, YPwmOutput 926
- isOnline, YPwmPowerSource 953
- isOnline, YQt 983
- isOnline, YRealTimeClock 1012
- isOnline, YRefFrame 1042
- isOnline, YRelay 1074
- isOnline, YSensor 1109
- isOnline, YServo 1141
- isOnline, YTemperature 1178
- isOnline, YTilt 1215
- isOnline, YVoc 1251
- isOnline, YVoltage 1287
- isOnline, YVSource 1320
- isOnline, YWakeUpMonitor 1347
- isOnline, YWakeUpSchedule 1382
- isOnline, YWatchdog 1420
- isOnline, YWireless 1456

J

- joinNetwork, YWireless 1457

L

- LightSensor 633
- lineTo, YDisplayLayer 399
- load, YAccelerometer 57
- load, YAnButton 94
- load, YCarbonDioxide 128
- load, YColorLed 157
- load, YCompass 191
- load, YCurrent 227
- load, YDataLogger 259
- load, YDigitalIO 321
- load, YDisplay 363
- load, YDualPower 428
- load, YFiles 452
- load, YGenericSensor 487
- load, YGyro 536
- load, YHubPort 566

- load, YHumidity 596
- load, YLed 624
- load, YLightSensor 657
- load, YMagnetometer 696
- load, YModule 744
- load, YNetwork 789
- load, YOsControl 823
- load, YPower 856
- load, YPressure 893
- load, YPwmOutput 927
- load, YPwmPowerSource 954
- load, YQt 984
- load, YRealTimeClock 1013
- load, YRefFrame 1043
- load, YRelay 1075
- load, YSensor 1110
- load, YServo 1142
- load, YTemperature 1179
- load, YTilt 1216
- load, YVoc 1252
- load, YVoltage 1288
- load, YVSource 1321
- load, YWakeUpMonitor 1348
- load, YWakeUpSchedule 1383
- load, YWatchdog 1421
- load, YWireless 1458
- loadCalibrationPoints, YAccelerometer 58
- loadCalibrationPoints, YCarbonDioxide 129
- loadCalibrationPoints, YCompass 192
- loadCalibrationPoints, YCurrent 228
- loadCalibrationPoints, YGenericSensor 488
- loadCalibrationPoints, YGyro 537
- loadCalibrationPoints, YHumidity 597
- loadCalibrationPoints, YLightSensor 658
- loadCalibrationPoints, YMagnetometer 697
- loadCalibrationPoints, YPower 857
- loadCalibrationPoints, YPressure 894
- loadCalibrationPoints, YQt 985
- loadCalibrationPoints, YSensor 1111
- loadCalibrationPoints, YTemperature 1180
- loadCalibrationPoints, YTilt 1217
- loadCalibrationPoints, YVoc 1253
- loadCalibrationPoints, YVoltage 1289
- loadMore, YDataSet 280

M

- Magnetometer 670
- Measured 709
- Module 5, 715
- more3DCalibration, YRefFrame 1044
- move, YServo 1143
- moveTo, YDisplayLayer 400

N

- Network 756
- newSequence, YDisplay 364
- nextAccelerometer, YAccelerometer 59
- nextAnButton, YAnButton 95

- nextCarbonDioxide, YCarbonDioxide 130
- nextColorLed, YColorLed 158
- nextCompass, YCompass 193
- nextCurrent, YCurrent 229
- nextDataLogger, YDataLogger 260
- nextDigitalIO, YDigitalIO 322
- nextDisplay, YDisplay 365
- nextDualPower, YDualPower 429
- nextFiles, YFiles 453
- nextGenericSensor, YGenericSensor 489
- nextGyro, YGyro 538
- nextHubPort, YHubPort 567
- nextHumidity, YHumidity 598
- nextLed, YLed 625
- nextLightSensor, YLightSensor 659
- nextMagnetometer, YMagnetometer 698
- nextModule, YModule 745
- nextNetwork, YNetwork 790
- nextOsControl, YOsControl 824
- nextPower, YPower 858
- nextPressure, YPressure 895
- nextPwmOutput, YPwmOutput 928
- nextPwmPowerSource, YPwmPowerSource 955
- nextQt, YQt 986
- nextRealTimeClock, YRealTimeClock 1014
- nextRefFrame, YRefFrame 1045
- nextRelay, YRelay 1076
- nextSensor, YSensor 1112
- nextServo, YServo 1144
- nextTemperature, YTemperature 1181
- nextTilt, YTilt 1218
- nextVoc, YVoc 1254
- nextVoltage, YVoltage 1290
- nextVSource, YVSource 1322
- nextWakeUpMonitor, YWakeUpMonitor 1349
- nextWakeUpSchedule, YWakeUpSchedule 1384
- nextWatchdog, YWatchdog 1422
- nextWireless, YWireless 1459

O

- Object 381

P

- pauseSequence, YDisplay 366
- ping, YNetwork 791
- playSequence, YDisplay 367
- Port 551
- Power 413, 830
- Preparation 3
- PreregisterHub, YAPI 20
- Pressure 870
- pulse, YDigitalIO 323
- pulse, YRelay 1077
- pulse, YVSource 1323
- pulse, YWatchdog 1423
- pulseDurationMove, YPwmOutput 929
- PwmPowerSource 941

Q

Quaternion 961

R

Real 997
reboot, YModule 746
Recorded 270
Reference 10, 1021
registerAnglesCallback, YGyro 539
RegisterDeviceArrivalCallback, YAPI 21
RegisterDeviceRemovalCallback, YAPI 22
RegisterHub, YAPI 23
RegisterHubDiscoveryCallback, YAPI 24
RegisterLogFunction, YAPI 25
registerQuaternionCallback, YGyro 540
registerTimedReportCallback, YAccelerometer 60
registerTimedReportCallback, YCarbonDioxide 131
registerTimedReportCallback, YCompass 194
registerTimedReportCallback, YCurrent 230
registerTimedReportCallback, YGenericSensor 490
registerTimedReportCallback, YGyro 541
registerTimedReportCallback, YHumidity 599
registerTimedReportCallback, YLightSensor 660
registerTimedReportCallback, YMagnetometer 699
registerTimedReportCallback, YPower 859
registerTimedReportCallback, YPressure 896
registerTimedReportCallback, YQt 987
registerTimedReportCallback, YSensor 1113
registerTimedReportCallback, YTemperature 1182
registerTimedReportCallback, YTilt 1219
registerTimedReportCallback, YVoc 1255
registerTimedReportCallback, YVoltage 1291
registerValueCallback, YAccelerometer 61
registerValueCallback, YAnButton 96
registerValueCallback, YCarbonDioxide 132
registerValueCallback, YColorLed 159
registerValueCallback, YCompass 195
registerValueCallback, YCurrent 231
registerValueCallback, YDataLogger 261
registerValueCallback, YDigitalIO 324
registerValueCallback, YDisplay 368
registerValueCallback, YDualPower 430
registerValueCallback, YFiles 454
registerValueCallback, YGenericSensor 491
registerValueCallback, YGyro 542
registerValueCallback, YHubPort 568
registerValueCallback, YHumidity 600
registerValueCallback, YLed 626
registerValueCallback, YLightSensor 661
registerValueCallback, YMagnetometer 700
registerValueCallback, YNetwork 792
registerValueCallback, YOsControl 825
registerValueCallback, YPower 860

registerValueCallback, YPressure 897
registerValueCallback, YPwmOutput 930
registerValueCallback, YPwmPowerSource 956
registerValueCallback, YQt 988
registerValueCallback, YRealTimeClock 1015
registerValueCallback, YRefFrame 1046
registerValueCallback, YRelay 1078
registerValueCallback, YSensor 1114
registerValueCallback, YServo 1145
registerValueCallback, YTemperature 1183
registerValueCallback, YTilt 1220
registerValueCallback, YVoc 1256
registerValueCallback, YVoltage 1292
registerValueCallback, YVSource 1324
registerValueCallback, YWakeUpMonitor 1350
registerValueCallback, YWakeUpSchedule 1385
registerValueCallback, YWatchdog 1424
registerValueCallback, YWireless 1460
Relay 1054
remove, YFiles 455
reset, YDisplayLayer 401
reset, YPower 861
resetAll, YDisplay 369
resetCounter, YAnButton 97
resetSleepCountDown, YWakeUpMonitor 1351
resetWatchdog, YWatchdog 1425
revertFromFlash, YModule 747
rgbMove, YColorLed 160

S

save3DCalibration, YRefFrame 1047
saveSequence, YDisplay 370
saveToFlash, YModule 748
selectColorPen, YDisplayLayer 402
selectEraser, YDisplayLayer 403
selectFont, YDisplayLayer 404
selectGrayPen, YDisplayLayer 405
Sensor 1087
Sequence 268, 270, 282
Servo 1123
set_adminPassword, YNetwork 793
set_analogCalibration, YAnButton 98
set_autoStart, YDataLogger 262
set_autoStart, YWatchdog 1426
set_beacon, YModule 749
set_bearing, YRefFrame 1048
set_bitDirection, YDigitalIO 325
set_bitOpenDrain, YDigitalIO 326
set_bitPolarity, YDigitalIO 327
set_bitState, YDigitalIO 328
set_blinking, YLed 627
set_brightness, YDisplay 371
set_calibrationMax, YAnButton 99
set_calibrationMin, YAnButton 100
set_callbackCredentials, YNetwork 794
set_callbackEncoding, YNetwork 795
set_callbackMaxDelay, YNetwork 796
set_callbackMethod, YNetwork 797
set_callbackMinDelay, YNetwork 798

set_callbackUrl, YNetwork 799
set_discoverable, YNetwork 800
set_dutyCycle, YPwmOutput 931
set_dutyCycleAtPowerOn, YPwmOutput 932
set_enabled, YDisplay 372
set_enabled, YHubPort 569
set_enabled, YPwmOutput 933
set_enabled, YServo 1146
set_enabledAtPowerOn, YPwmOutput 934
set_enabledAtPowerOn, YServo 1147
set_frequency, YPwmOutput 935
set_highestValue, YAccelerometer 62
set_highestValue, YCarbonDioxide 133
set_highestValue, YCompass 196
set_highestValue, YCurrent 232
set_highestValue, YGenericSensor 492
set_highestValue, YGyro 543
set_highestValue, YHumidity 601
set_highestValue, YLightSensor 662
set_highestValue, YMagnetometer 701
set_highestValue, YPower 862
set_highestValue, YPressure 898
set_highestValue, YQt 989
set_highestValue, YSensor 1115
set_highestValue, YTemperature 1184
set_highestValue, YTilt 1221
set_highestValue, YVoc 1257
set_highestValue, YVoltage 1293
set_hours, YWakeUpSchedule 1386
set_hslColor, YColorLed 161
set_logFrequency, YAccelerometer 63
set_logFrequency, YCarbonDioxide 134
set_logFrequency, YCompass 197
set_logFrequency, YCurrent 233
set_logFrequency, YGenericSensor 493
set_logFrequency, YGyro 544
set_logFrequency, YHumidity 602
set_logFrequency, YLightSensor 663
set_logFrequency, YMagnetometer 702
set_logFrequency, YPower 863
set_logFrequency, YPressure 899
set_logFrequency, YQt 990
set_logFrequency, YSensor 1116
set_logFrequency, YTemperature 1185
set_logFrequency, YTilt 1222
set_logFrequency, YVoc 1258
set_logFrequency, YVoltage 1294
set_logicalName, YAccelerometer 64
set_logicalName, YAnButton 101
set_logicalName, YCarbonDioxide 135
set_logicalName, YColorLed 162
set_logicalName, YCompass 198
set_logicalName, YCurrent 234
set_logicalName, YDataLogger 263
set_logicalName, YDigitalIO 329
set_logicalName, YDisplay 373
set_logicalName, YDualPower 431
set_logicalName, YFiles 456
set_logicalName, YGenericSensor 494

set_logicalName, YGyro 545
set_logicalName, YHubPort 570
set_logicalName, YHumidity 603
set_logicalName, YLed 628
set_logicalName, YLightSensor 664
set_logicalName, YMagnetometer 703
set_logicalName, YModule 750
set_logicalName, YNetwork 801
set_logicalName, YOsControl 826
set_logicalName, YPower 864
set_logicalName, YPressure 900
set_logicalName, YPwmOutput 936
set_logicalName, YPwmPowerSource 957
set_logicalName, YQt 991
set_logicalName, YRealTimeClock 1016
set_logicalName, YRefFrame 1049
set_logicalName, YRelay 1079
set_logicalName, YSensor 1117
set_logicalName, YServo 1148
set_logicalName, YTemperature 1186
set_logicalName, YTilt 1223
set_logicalName, YVoc 1259
set_logicalName, YVoltage 1295
set_logicalName, YVSource 1325
set_logicalName, YWakeUpMonitor 1352
set_logicalName, YWakeUpSchedule 1387
set_logicalName, YWatchdog 1427
set_logicalName, YWireless 1461
set_lowestValue, YAccelerometer 65
set_lowestValue, YCarbonDioxide 136
set_lowestValue, YCompass 199
set_lowestValue, YCurrent 235
set_lowestValue, YGenericSensor 495
set_lowestValue, YGyro 546
set_lowestValue, YHumidity 604
set_lowestValue, YLightSensor 665
set_lowestValue, YMagnetometer 704
set_lowestValue, YPower 865
set_lowestValue, YPressure 901
set_lowestValue, YQt 992
set_lowestValue, YSensor 1118
set_lowestValue, YTemperature 1187
set_lowestValue, YTilt 1224
set_lowestValue, YVoc 1260
set_lowestValue, YVoltage 1296
set_luminosity, YLed 629
set_luminosity, YModule 751
set_maxTimeOnStateA, YRelay 1080
set_maxTimeOnStateA, YWatchdog 1428
set_maxTimeOnStateB, YRelay 1081
set_maxTimeOnStateB, YWatchdog 1429
set_minutes, YWakeUpSchedule 1388
set_minutesA, YWakeUpSchedule 1389
set_minutesB, YWakeUpSchedule 1390
set_monthDays, YWakeUpSchedule 1391
set_months, YWakeUpSchedule 1392
set_mountPosition, YRefFrame 1050
set_neutral, YServo 1149
set_nextWakeUp, YWakeUpMonitor 1353

set_orientation, YDisplay 374
 set_output, YRelay 1082
 set_output, YWatchdog 1430
 set_outputVoltage, YDigitalIO 330
 set_period, YPwmOutput 937
 set_portDirection, YDigitalIO 331
 set_portOpenDrain, YDigitalIO 332
 set_portPolarity, YDigitalIO 333
 set_portState, YDigitalIO 334
 set_position, YServo 1150
 set_positionAtPowerOn, YServo 1151
 set_power, YLed 630
 set_powerControl, YDualPower 432
 set_powerDuration, YWakeUpMonitor 1354
 set_powerMode, YPwmPowerSource 958
 set_primaryDNS, YNetwork 802
 set_pulseDuration, YPwmOutput 938
 set_range, YServo 1152
 set_recording, YDataLogger 264
 set_reportFrequency, YAccelerometer 66
 set_reportFrequency, YCarbonDioxide 137
 set_reportFrequency, YCompass 200
 set_reportFrequency, YCurrent 236
 set_reportFrequency, YGenericSensor 496
 set_reportFrequency, YGyro 547
 set_reportFrequency, YHumidity 605
 set_reportFrequency, YLightSensor 666
 set_reportFrequency, YMagnetometer 705
 set_reportFrequency, YPower 866
 set_reportFrequency, YPressure 902
 set_reportFrequency, YQt 993
 set_reportFrequency, YSensor 1119
 set_reportFrequency, YTemperature 1188
 set_reportFrequency, YTilt 1225
 set_reportFrequency, YVoc 1261
 set_reportFrequency, YVoltage 1297
 set_resolution, YAccelerometer 67
 set_resolution, YCarbonDioxide 138
 set_resolution, YCompass 201
 set_resolution, YCurrent 237
 set_resolution, YGenericSensor 497
 set_resolution, YGyro 548
 set_resolution, YHumidity 606
 set_resolution, YLightSensor 667
 set_resolution, YMagnetometer 706
 set_resolution, YPower 867
 set_resolution, YPressure 903
 set_resolution, YQt 994
 set_resolution, YSensor 1120
 set_resolution, YTemperature 1189
 set_resolution, YTilt 1226
 set_resolution, YVoc 1262
 set_resolution, YVoltage 1298
 set_rgbColor, YColorLed 163
 set_rgbColorAtPowerOn, YColorLed 164
 set_running, YWatchdog 1431
 set_secondaryDNS, YNetwork 803
 set_sensitivity, YAnButton 102
 set_sensorType, YTemperature 1190
 set_signalRange, YGenericSensor 498
 set_sleepCountdown, YWakeUpMonitor 1355
 set_startupSeq, YDisplay 375
 set_state, YRelay 1083
 set_state, YWatchdog 1432
 set_stateAtPowerOn, YRelay 1084
 set_stateAtPowerOn, YWatchdog 1433
 set_timeUTC, YDataLogger 265
 set_triggerDelay, YWatchdog 1434
 set_triggerDuration, YWatchdog 1435
 set_unit, YGenericSensor 499
 set_unixTime, YRealTimeClock 1017
 set_usbBandwidth, YModule 752
 set_userData, YAccelerometer 68
 set_userData, YAnButton 103
 set_userData, YCarbonDioxide 139
 set_userData, YColorLed 165
 set_userData, YCompass 202
 set_userData, YCurrent 238
 set_userData, YDataLogger 266
 set_userData, YDigitalIO 335
 set_userData, YDisplay 376
 set_userData, YDualPower 433
 set_userData, YFiles 457
 set_userData, YGenericSensor 500
 set_userData, YGyro 549
 set_userData, YHubPort 571
 set_userData, YHumidity 607
 set_userData, YLed 631
 set_userData, YLightSensor 668
 set_userData, YMagnetometer 707
 set_userData, YModule 753
 set_userData, YNetwork 804
 set_userData, YOsControl 827
 set_userData, YPower 868
 set_userData, YPressure 904
 set_userData, YPwmOutput 939
 set_userData, YPwmPowerSource 959
 set_userData, YQt 995
 set_userData, YRealTimeClock 1018
 set_userData, YRefFrame 1051
 set_userData, YRelay 1085
 set_userData, YSensor 1121
 set_userData, YServo 1153
 set_userData, YTemperature 1191
 set_userData, YTilt 1227
 set_userData, YVoc 1263
 set_userData, YVoltage 1299
 set_userData, YVSource 1326
 set_userData, YWakeUpMonitor 1356
 set_userData, YWakeUpSchedule 1393
 set_userData, YWatchdog 1436
 set_userData, YWireless 1462
 set_userPassword, YNetwork 805
 set_utcOffset, YRealTimeClock 1019
 set_valueRange, YGenericSensor 501
 set_voltage, YVSource 1327
 set_weekDays, YWakeUpSchedule 1394
 set_wwwWatchdogDelay, YNetwork 806

setAntialiasingMode, YDisplayLayer 406
setConsoleBackground, YDisplayLayer 407
setConsoleMargins, YDisplayLayer 408
setConsoleWordWrap, YDisplayLayer 409
setLayerPosition, YDisplayLayer 410
shutdown, YOsControl 828
Sleep, YAPI 26
sleep, YWakeUpMonitor 1357
sleepFor, YWakeUpMonitor 1358
sleepUntil, YWakeUpMonitor 1359
Source 1301
start3DCalibration, YRefFrame 1052
stopSequence, YDisplay 377
Supply 413
swapLayerContent, YDisplay 378

T

Temperature 1155
Tilt 1193
Time 997
toggle_bitState, YDigitalIO 336
triggerFirmwareUpdate, YModule 754
TriggerHubDiscovery, YAPI 27

U

Unformatted 282
unhide, YDisplayLayer 411
UnregisterHub, YAPI 28
UpdateDeviceList, YAPI 29
upload, YDisplay 379
upload, YFiles 458
useDHCP, YNetwork 807
useStaticIP, YNetwork 808

V

Value 709
Voltage 1265, 1301
voltageMove, YVSource 1328

W

wakeUp, YWakeUpMonitor 1360
WakeUpMonitor 1330
WakeUpSchedule 1362
Watchdog 1396
Wireless 1438

Y

YAccelerometer 33-68
YAnButton 72-103
YAPI 12-29
YCarbonDioxide 107-139
yCheckLogicalName 12
YColorLed 142-165
YCompass 169-202
YCurrent 206-238
YDataLogger 241-266

YDataRun 268
YDataSet 271-280
YDataStream 283-295
YDigitalIO 299-336
yDisableExceptions 13
YDisplay 340-379
YDisplayLayer 382-411
YDualPower 414-433
yEnableExceptions 14
YFiles 436-458
yFindAccelerometer 33
yFindAnButton 72
yFindCarbonDioxide 107
yFindColorLed 142
yFindCompass 169
yFindCurrent 206
yFindDataLogger 241
yFindDigitalIO 299
yFindDisplay 340
yFindDualPower 414
yFindFiles 436
yFindGenericSensor 462
yFindGyro 505
yFindHubPort 552
yFindHumidity 575
yFindLed 610
yFindLightSensor 635
yFindMagnetometer 672
yFindModule 717
yFindNetwork 759
yFindOsControl 811
yFindPower 832
yFindPressure 872
yFindPwmOutput 908
yFindPwmPowerSource 942
yFindQt 963
yFindRealTimeClock 998
yFindRefFrame 1023
yFindRelay 1056
yFindSensor 1089
yFindServo 1125
yFindTemperature 1157
yFindTilt 1195
yFindVoc 1231
yFindVoltage 1267
yFindVSource 1302
yFindWakeUpMonitor 1332
yFindWakeUpSchedule 1364
yFindWatchdog 1398
yFindWireless 1439
yFirstAccelerometer 34
yFirstAnButton 73
yFirstCarbonDioxide 108
yFirstColorLed 143
yFirstCompass 170
yFirstCurrent 207
yFirstDataLogger 242
yFirstDigitalIO 300
yFirstDisplay 341

yFirstDualPower 415
yFirstFiles 437
yFirstGenericSensor 463
yFirstGyro 506
yFirstHubPort 553
yFirstHumidity 576
yFirstLed 611
yFirstLightSensor 636
yFirstMagnetometer 673
yFirstModule 718
yFirstNetwork 760
yFirstOsControl 812
yFirstPower 833
yFirstPressure 873
yFirstPwmOutput 909
yFirstPwmPowerSource 943
yFirstQt 964
yFirstRealTimeClock 999
yFirstRefFrame 1024
yFirstRelay 1057
yFirstSensor 1090
yFirstServo 1126
yFirstTemperature 1158
yFirstTilt 1196
yFirstVoc 1232
yFirstVoltage 1268
yFirstVSource 1303
yFirstWakeUpMonitor 1333
yFirstWakeUpSchedule 1365
yFirstWatchdog 1399
yFirstWireless 1440
yFreeAPI 15
YGenericSensor 462-501
yGetAPIVersion 16
yGetTickCount 17
YGyro 505-549
yHandleEvents 18
YHubPort 552-571
YHumidity 575-607

yInitAPI 19
YLed 610-631
YLightSensor 635-668
YMagnetometer 672-707
YMeasure 709-713
YModule 717-754
YNetwork 759-808
Yocto-Demo 3
Yocto-hub 551
YOsControl 811-828
YPower 832-868
yPreregisterHub 20
YPressure 872-904
YPwmOutput 908-939
YPwmPowerSource 942-959
YQt 963-995
YRealTimeClock 998-1019
YRefFrame 1023-1052
yRegisterDeviceArrivalCallback 21
yRegisterDeviceRemovalCallback 22
yRegisterHub 23
yRegisterHubDiscoveryCallback 24
yRegisterLogFunction 25
YRelay 1056-1085
YSensor 1089-1121
YServo 1125-1153
ySleep 26
YTemperature 1157-1191
YTilt 1195-1227
yTriggerHubDiscovery 27
yUnregisterHub 28
yUpdateDeviceList 29
YVoc 1231-1263
YVoltage 1267-1299
YVSource 1302-1328
YWakeUpMonitor 1332-1360
YWakeUpSchedule 1364-1394
YWatchdog 1398-1436
YWireless 1439-1462